TASK: PA18
CDRL: A023
26 February 1994

Report on Logical Foundations

Informal Technical Data

STARS-AC-A023/005/00
26 February 1994

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	26 Feb 1994	Informal Technical Report

4. TITLE AND SUBTITLE

Report on Logical Foundations

5. FUNDING NUMBERS

F19628-93-C-0130

6. AUTHOR(S)

ORA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Unisys Corporation
12010 Sunrise Valley Drive
Reston, VA 22091

8. PERFORMING ORGANIZATION REPORT NUMBER

STARS-AC-A023/005/00

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of the Air Force
Headquarters ESC
Hanscom, AFB, MA 01731-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

A023

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution "A"

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Penelope Ada proof editor allows a user to incrementally develop provably correct Ada programs. The current version of the Penelope system is formally based on a predicate transformer semantics for sequential Ada. The purpose of this work is to provide the mathematical foundations for extending Penelope to Ada tasking programs.

14. SUBJECT TERMS

15. NUMBER OF PAGES
77

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

TASK: PA18
CDRL: A023
26 February 1994

INFORMAL TECHNICAL REPORT

For

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS)

Report on Logical Foundations

STARS-AC-A023/005/00
26 February 1994

Data Type: Informal Technical Data

CONTRACT NO. F19628-93-C-0130

Prepared for:

Electronic Systems Center
Air Force Materiel Command, USAF
Hanscom AFB, MA 01731-2816

Prepared by:

Odyssey Research Associates
under contract to
Unisys Corporation
12010 Sunrise Valley Drive
Reston, VA 22091

TASK: PA18
CDRL: A023
26 February 1994

Data Reference: STARS-AC-A023/005/00
INFORMAL TECHNICAL REPORT
Report on Logical
Foundations

Distribution Statement "A"
per DoD Directive 5230.24
Authorized for public release; Distribution is unlimited.

This document, developed under the Software Technology for Adaptable, Reliable Systems (STARS) program, is approved for release under Distribution "A" of the Scientific and Technical Information Program Classification Scheme (DoD Directive 5230.24) unless otherwise indicated. Sponsored by the U.S. Advanced Research Projects Agency (ARPA) under contract F19628-93-C-0130, the STARS program is supported by the military services, SEI, and MITRE, with the U.S. Air Force as the executive contracting agent. The information identified herein is subject to change. For further information, contact the authors at the following mailer address: delivery@stars.reston.paramax.com

TASK: PA18
CDRL: A023
26 February 1994

Data Reference: STARS-AC-A023/005/00
INFORMAL TECHNICAL REPORT
Report on Logical
Foundations

Principal Author(s):

Wolfgang Polak *Date*

Approvals:

Program Manager *Teri F. Payton* *Date*

(Signatures on File)

Contents

1 Introduction **1**
- 1.1 Goals and Objectives . . . 1
- 1.2 Specifying Concurrent Systems . . . 3
- 1.3 Process Logic . . . 4
- 1.4 Ada Verification . . . 10
  - 1.4.1 Principles . . . 11
  - 1.4.2 Restrictions . . . 12
    - 1.4.2.1 Termination . . . 12
    - 1.4.2.2 Timing . . . 13
    - 1.4.2.3 Priorities . . . 13
    - 1.4.2.4 Task Status . . . 13
    - 1.4.2.5 Abort statement . . . 13
    - 1.4.2.6 Entry queues . . . 14
    - 1.4.2.7 Global variables . . . 14
- 1.5 Related Work . . . 14
  - 1.5.1 Process Algebra . . . 14
  - 1.5.2 Other Approaches . . . 15
    - 1.5.2.1 Modal logic . . . 15
    - 1.5.2.2 Axiomatic Methods . . . 16
    - 1.5.2.3 History Sequences . . . 16
  - 1.5.3 Other Specification Languages . . . 17
    - 1.5.3.1 LOTOS . . . 17
    - 1.5.3.2 RSL . . . 18
  - 1.5.4 Support Tools . . . 18
- 1.6 Organization . . . 19

2 Formal Basis **19**
- 2.1 Notation . . . 19
- 2.2 Domains . . . 19
- 2.3 First-order Logic . . . 21
- 2.4 Processes . . . 22
- 2.5 Operational Semantics . . . 23
- 2.6 Process Equivalence . . . 26
- 2.7 The "don't care" Process . . . 28
- 2.8 Proof Theory . . . 28
  - 2.8.1 Equations . . . 28
  - 2.8.2 Approximation . . . 32
  - 2.8.3 Example Proofs . . . 32
- 2.9 Denotational Semantics . . . 34
  - 2.9.1 Process Semantics . . . 37

3 Semantics and Verification **38**
- 3.1 Concepts . . . 38

3.1.1 Predicate Transformers . . . 38
3.1.2 Application to Concurrency . . . 39
3.1.3 A Special Case: Sequential Programs . . . 40
3.2 Ada Tasking . . . 42
3.2.1 Task Declarations . . . 43
3.2.2 Task Activation . . . 44
3.2.3 Task Termination . . . 45
3.2.4 Entry Calls . . . 46
3.2.5 Entries and Accept Statements . . . 47
3.2.6 Delay Statements . . . 47
3.2.7 Select Statements . . . 47
3.2.8 Conditional Entry Calls . . . 48
3.2.9 Abort Statements . . . 48
3.3 Annotations and Proofs . . . 48
3.3.1 Invariants in Concurrent Programs . . . 48
3.3.2 Procedure Annotations . . . 50
3.3.3 Abstraction and Action Refinement . . . 50

4 Examples **51**
4.1 A Buffer Task . . . 51
4.1.1 The Program . . . 51
4.1.2 Annotations . . . 52
4.1.3 Verification Conditions . . . 53
4.2 Multi-Set Partitioning . . . 55
4.2.1 Design . . . 55
4.2.2 The Program . . . 57
4.2.3 Verification Conditions . . . 58
4.2.4 A Calling Environment . . . 59
4.3 Matrix Multiplication . . . 62
4.3.1 The Program . . . 62
4.3.2 Problem Areas . . . 67

5 Conclusions **68**

6 Bibliography **70**

1 Introduction

1.1 Goals and Objectives

The Penelope Ada proof editor allows a user to incrementally develop provably correct Ada programs. The current version of the Penelope system is formally based on a predicate transformer semantics for sequential Ada. The purpose of this work is to provide the mathematical foundations for extending Penelope to Ada tasking programs. Several issues need to be addressed to this end.

1. A suitable logical formalism needs to be defined within which it is possible to reason about concurrent programs. In the case of sequential programs first-order logic was sufficient for this purpose.
2. A specification and an annotation language must be defined for stating properties of concurrent programs. In the case of sequential programs input/output conditions and loop invariants are used as specifications.
3. A method for defining the semantics of Ada needs to be devised. Given a program and a specification, it must be possible to derive conditions under which the program will satisfy its specifications. This step corresponds to the generation of verification conditions in the sequential case.
4. An effective proof procedure must be found for showing that the that correctness conditions generated for a program are true. If first-order logic is used, then a first-order theorem prover will suffice, in the concurrent case a theorem prover for a richer language will be needed.
5. Finally, a new methodology for the systematic development of correct tasking programs needs to be developed. This includes ways to formally express problem specification, to find proper program annotations and so on.

This document addresses primarily the first three questions. While an appropriate notion of proof is formally defined within our mathematical framework, issues of the engineering of a practical theorem prover are not discussed. The use of the new formalism for proving the correctness of tasking programs is demonstrated with several programming examples. But the development of a verification methodology requires more extensive experience.

There are a number of desiderata for the formalism.

1. The specification method must be compositional and must support abstractions. This is to say, that the meaning of a concurrent program can be determined from the meaning of its parts and that these parts can be separately specified and proved correct.

   Compositionality is important so the one can break down complex proofs into smaller, manageable ones. Compositionality is difficult to achieve for concurrent programs since their semantics depends on the possible interaction of all processes.

2. Program annotations must express the intended behavior in a natural way. The specifications of a program's parts should correspond to notions of the designer and programmer.

3. There must be reasonably efficient proof procedures for the formal correctness statements. Given that fully automated proofs are not likely (and impossible in general), it is important that the formulas arising during proofs are meaningful to the human verifier and that they can be related to the program text.

The approach taken here is based on the concepts of *process algebra*, an abstract, algebraic description of the observable behavior of processes [87, 18, 57]. Process algebra is compositional, leads to fairly natural specifications, and has well understood proof procedures. The technique is fairly mature and other formal specification languages such as LOTOS ([22]) and the RAISE specification language ([44]) are also based on process algebra.

The contribution of this work is twofold. First, it defines *process logic* which combines first-order logic with process algebra[1]. The former allows the specification of the state of a computation while the latter describes the possible sequences of observable events. Secondly, a technique, based on predicate transformers, is defined for using process logic to specify and verify concurrent programs.

Using process logic it is possible to express the communication behavior of a code fragment in terms of pre- and post-processes. I.e. process terms are used in place of pre- and post-conditions. The method is compositional in the sense that the pre- and post-process of a composite language construct are defined in terms of the pre- and post-processes of its constituents.

Even though process logic can express divergence, termination, and deadlock, the proposed specification technique deals only with partial correctness. The reason is that as in the sequential case, loop invariants and pre- and post-processes are used to reason about iteration and recursion.

The technique assumes a message passing semantics of concurrency. I.e., the computational model assumes a set of processes with local state that communicate through messages. In particular, the method does not deal directly with shared memory concurrency. Of course, shared memory accesses can always be viewed as communications with a central shared memory process. The method is applied to Ada by specifying predicate transformers for Ada tasking construct. But the definition of process logic is independent of Ada and can be applied to other languages.

The method described here satisfies the above criteria to large degree. Predicate transformers are a natural mechanism for several reasons. Program specifications can be expressed through pre- and post-processes on code fragments. Program verification proceeds in a manner

[1]The term *process logic* is sometimes used (e.g. in [94]) to refer to Hennessy-Milner logic ([55]). The use of the term in the present context has historic reasons since early version of this work were based on a modal logic approach.

familiar from sequential verification: For a given program with annotations the verifier will generate verification conditions which are shown to be valid by a theorem prover.

1.2 Specifying Concurrent Systems

Concurrency is used on a number of levels in software systems. Different usages may require different formal models. It is therefore important to define the kinds of applications and correctness properties that we are interested in.

One class of problems that are of no interest here involves low-level concurrency such as the implementation of mutual exclusion in terms of shared variables with atomic assignment, or the correctness proof of an on-the-fly garbage collector ([39]). Rather, the problems of interest involve concurrent processes that communicate with each other through message passing. It is assumed that on this level suitable process abstractions are available. Issues of the correct implementation of processes abstraction, of the fairness of the system scheduler, and so on are of no concern here. The view taken here is very broad and includes all components of a system. These need not be software artifacts but may involve hardware devices. The correctness of a software component of such a larger system needs to be proved in the context of its environment. The Ada tasking model supports this more abstract view where devices interact with Ada tasks through interrupts mapped to entry calls.

The typical system of interest might be a reactive system that responds to a set of possible stimuli by certain actions. For example, the control system of a reactor receives inputs from a number of pressure and temperature sensors and has to properly respond by issuing appropriate signals to control valves, sound alarms and so on. Within our formal model it is possible to describe the behavior of such a control system as well as the reactor itself. It is then possible to formally evaluate their mutual interaction. Eventually, the control system or significant parts of it will be implemented in software in a suitable high-level language (e.g. Ada).

Most of the systems specified are likely to be non-terminating. In fact, termination of a control system might be a fatal error. Rather, the kind of termination properties one is interested in are called *liveness properties* (e.g. [72, 7]). Liveness means that the program is guaranteed to make progress and will eventually respond. By contrast, *safety properties* guarantee that the program, if it responds, will respond properly. (See [1] for a topological definition of safety and liveness.)

In general, liveness may be violated by non-termination, either due to non-termination of a sequential program fragment or infinite internal communication. The former situation requires termination proofs of all sequential fragments. The latter case also amounts to termination: on some level of abstraction it is not observable whether internal computation involves communication or not. The present work is only concerned with safety properties, i.e., the causal relation between events and with the correctness of associated data. This can be remedied, however, by using standard well-foundedness arguments to prove termination of sequential program parts.

In practice one is not only concerned with termination but also with the actual time it takes to respond to certain events. Timing constraints are not treated here. There are several reasons for omitting timing consideration. One important problem is that the timing behavior of a program is not a property of the source code but depends on a large number of other factors such as compilation techniques, processor performance, distribution and so on. At least in principle, process algebra can be extended to deal with real-time (see e.g. [16]). Henzinger ([56]) discusses the notion of liveness in the context of timing constraints.

Another property of concurrent programs is fairness. An execution is not fair if some service request is ignored for ever. This situation may arise if the program performs useful work on other tasks. Process logic cannot express fairness in this abstract form. This is no big loss in practice, since the abstract fairness property only guarantees that service will eventually, after arbitrary finite delay, be provided. A statement that says that at most a certain fixed number of other events happen before service is provided is much more useful for real application. But such a statement of bounded delay (not in time but in terms of the number of observable actions) can be expressed in process logic.

One key question, of course, is how properties of a concurrent program can be specified in a natural manner. Pre- and postconditions have been used successfully for the specification of sequential programs. Even though such techniques use internal annotations of loops and local subprograms, the specifications are extensional. This is to say any program that satisfies the prescribed input/output behavior satisfies the specifications. For sequential programs the observable behavior is fully determined by by the relation between inputs and outputs of the program.

The process logic approach uses pre- and post-processes that express both the expected computation states as well as the communications of the program fragment with its environment. The description of the possible communication behavior must be exact in the sense that it captures all possible behaviors. The reason is compositionality: only if all possible behaviors of two processes are known is it possible to determine their interaction.

1.3 Process Logic

In the following we give an informal presentation of process logic. The purpose is to motivate the choice of constructs and make it easier to follow the formal definition given in section 2.

First, the terminology may need some clarification. Often the notion of a process is associated with a physical (or at least abstract) agent. Instead, in the present context the word process refers to a sequence of actions being performed. For example, two processes performed concurrently constitute a new process. The exact nature of the agents (e.g. Ada tasks) that realize a process is immaterial. Observationally, the execution of multiple agents constitutes a single process. In section 3.1 the semantic definition of Ada tasking will formally establish the relation between a set of tasks and their meaning (a process).

Primitive processes. Complex processes are constructed from simple primitive processes by a number of operators that modify or combine processes. There are three primitive processes, ϵ, δ, and Ω. ϵ is the terminated process that will not perform any further actions. δ is the deadlocked process; it will not perform any further actions but has not terminated. The distinction between deadlock and normal termination is very important in the computational setting. There is an observable difference between the two processes if they are sequentially composed with another process (see below). Process Ω indicates the divergent process, i.e. a process that continually performs internal computations but will not perform any observable action. Depending on the physical environment, a deadlock may or may not be observably different from Ω. For instance, a deadlocked program will appear to be non-terminating but the fact that it does not use resources may be observable.

Actions If p is a process, $a \cdot p$ is a process that consists of performing the action a followed by process p. Intuitively, there can be a number of different kinds of actions such as sending or receiving a message. But in the formalism they are all described uniformly. Some actions require the cooperation of an observer, which may be another process or the environment (e.g. user, physical device etc.). Consider a very simple process

$$\text{PushButton?} \cdot \text{LightOn!} \cdot \epsilon$$

that waits for a button to be pushed and will then turn on a light and terminate. There is a difference between the event PushButton? and the event LightOn!. The former is caused by the user action PushButton! and is sensed by the process, while the latter is caused by the process and received by the environment as LightOn?. Both actions are synchronous and require the cooperation of two processes (one of which may be the environment).

Alternatives If p_1 and p_2 are processes, then $p_1 + p_2$ is a process that can choose between p_1 and p_2. This choice is non-deterministic. But in the case where the initial actions of p_1 and p_2 require cooperation, such as in

$$(\alpha! \cdot p_1) + (\beta! \cdot p_2)$$

the behavior of the process depends on the environment (e.g. whether the action α? or the action β? becomes possible first).

Recursion The processes that can be written so far are all finite and not very interesting. To define infinite or potentially infinite processes, recursive definitions of processes are introduced. For instance, a semaphore can be defined recursively as

$$\text{sem} ::= P? \cdot V? \cdot \text{sem}$$

This process accepts an infinite sequence of alternating $P!$ and $V!$ operations. Strictly speaking, such a recursive definition is merely a shorthand for recursion operator.

As a more realistic example consider a dish washer. It has a door that can be opened and closed, a start button, and a light that indicates that a wash cycle is complete. The dish washer process can be described by

$$\begin{aligned} \text{DishWasher} &::= \text{close?} \cdot (\text{start?} \cdot \text{Cycle} + \text{open?} \cdot \text{DishWasher}) \\ \text{Cycle} &::= (\text{open?} \cdot \text{close?} \cdot \text{Cycle}) \\ &\quad + (\text{LightOn!} \cdot \text{open?} \cdot \text{LightOff!} \cdot \text{DishWasher}) \end{aligned}$$

Internal action Suppose one wants to define a process that makes a nondeterministic choice and then engages in either action $\alpha!$ or action $\beta!$. The process $\alpha! \cdot \epsilon + \beta! \cdot \epsilon$ does *not* describe such a process since its behavior depends on the kind of communications offered by the environment. To define true non-determinism a new process term is introduced: $\tau \cdot p$ is a process that can perform some internal computation and will then behave like p. Whether or not τ occurs cannot be affected by the environment. Action τ is the only action that is not observable.

Consider the process

$$\text{Random} ::= (\tau \cdot \alpha! \cdot \text{Random}) + (\tau \cdot \beta! \cdot \text{Random})$$

Process *Random* can make an initial choice between two alternatives. Because either choice leads to internal computations, the choice itself cannot be influenced by the environment. For example, a faulty dish washer might be characterized by

$$\text{DishWasher} ::= \text{close?} \cdot (\tau \cdot \text{start?} \cdot \text{Cycle} + \tau \cdot \text{open?} \cdot \text{DishWasher})$$

meaning that, depending on some inner non-determinism, the machine allows the user to either push the start button or to open the door, but not both.

Value passing So far, the language can describe potentially infinite processes that engage in a finite set of actions. But these actions are atomic and no data communication is associated with their occurrence. This is not a problem in principle since the communication of arbitrary data could be modeled by a sequence of 0 and 1 actions representing individual bits. But clearly such an encoding would be highly inconvenient for describing practical processes.

Instead, in order to describe communications of data to and from a process, the notation is extended to $\alpha!v \cdot p$ and $\alpha?x \cdot p$. Here α is a *label* or name of a communication channel. The term $\alpha!v \cdot p$ denotes a process that will send the value v via channel α and will then behave like p. The term $\alpha?x \cdot p$ is a process that will receive some value via channel α and will bind it to variable x in p. The conceptual model is one where a send action is possibly only when some other process performs a matching receive action with the same label. The notation $\alpha! \cdot p$ and $\alpha? \cdot p$ is just the the special case of sending and receiving an empty message.

A simple process might be

$$\begin{aligned} \text{Calc} ::= \ & \text{input?}x \cdot \text{input?}y \cdot \\ & (\text{plus?} \cdot \text{output!}(x+y) \cdot \text{Calc} + \text{minus?} \cdot \text{output!}(x-y) \cdot \text{Calc}) \end{aligned}$$

Conditionals Once value passing can be described, one may want to define processes whose future behavior depends on the values received. So far the language provides no mechanism to express such dependencies. The problem is addressed by the following guard construct: If p and q are processes and A is a predicate, then if A then p else q is a new process. if A then p else q behaves like p if the predicate A is true and behaves like q otherwise.

Using conditionals, processes whose behavior depends on values received can be described. For example,

$$p ::= \text{input?}x \cdot \text{if } x \bmod 2 = 0 \text{ then}(\text{even!} \cdot p) \text{ else}(\text{odd!} \cdot p)$$

Extending recursion The kind of recursive processes that can be defined so far are cyclic processes whose behavior is independent of the local state, i.e. the recursive process will behave identically for every cycle. In a more general setting one may want to describe cyclic processes whose behavior depends on some local state that may be different for every iteration. This problem is addressed by introducing functions from states to processes and allowing recursive definitions of such functions. For example a potentially infinite buffer can be described as follows:

$$\begin{aligned} b(\text{cont}) ::= \ & \text{put?}x \cdot b(\text{cont}\&\langle x \rangle) \\ & + \text{ if } \neg\text{empty}(\text{cont}) \text{ then get!fst}(\text{cont}) \cdot b(\text{rest}(\text{cont})) \text{ else } \delta \end{aligned}$$

Here *cont* is a sequence denoting the current content of the buffer. $\langle x \rangle$ denotes a singleton sequence and & denotes sequence concatenation. A new empty buffer process is given by the term $b(\langle\rangle)$. Again, the notation is strictly shorthand for a more complex recursion operator.

Parallel composition The *producer* process

$$p ::= \text{put!}c_0 \cdot p$$

generates an infinite number of constants c_0. Process p can be executed in parallel with a buffer $b(\langle\rangle)$. This new process is written as $p \mid b(\langle\rangle)$. One might expect $p \mid b(\langle\rangle)$ to be a process that accepts an infinite number of *get* communications, each one communicating the value c_0. But this is not quite the case, because both processes can still communicate with the environment. This means, for instance, that $p \mid b(\langle\rangle)$ can accept a message sent to *put* and place the received data in the buffer.

To be precise $p \mid b(\langle\rangle)$ will be a process $bp(\langle\rangle)$ where

$$\begin{aligned} bp(\text{cont}) ::= \ & \text{put}?x \cdot bp(\text{cont} \cdot \langle x \rangle) \\ & + \ \textsf{if} \neg\text{empty}(\text{cont}) \ \textsf{then} \ \text{get}!\text{fst}(\text{cont}) \cdot bp(\text{rest}(\text{cont})) \ \textsf{else} \ \delta \\ & + \ \text{put}!c_0 \cdot bp(\text{cont}) \\ & + \ \tau \cdot bp(\text{cont} \cdot \langle c_0 \rangle) \end{aligned}$$

Section 2.8.1 defines rules for formally establishing the above equivalence.

It is important to guarantee that processes do not share state. This can be ensured by the requiring that two terms can only be parallel composed if their respective sets of free variables are disjoint.

Hiding labels A new operation is needed in order to describe a process that has local labels that are not externally observable. For instance one may want to define a process just like $p \mid b(\langle\rangle)$ that does not engage in any send action to *put*. Such a process is defined by the *hiding* operator.

$$\partial\{\text{put}\}.(p \mid b(\langle\rangle))$$

In general $\partial H.p$ denotes a process that behaves like p except that no communications via α are possible for any $\alpha \in H$. Applied to the example we get $\partial\{\text{put}\}.(p \mid b(\langle\rangle)) = bp'(\langle\rangle)$ where

$$\begin{aligned} bp'(\text{cont}) ::= \ & \textsf{if} \neg\text{empty}(\text{cont}) \ \textsf{then} \ \text{get}!\text{fst}(\text{cont}) \cdot bp'(\text{rest}(\text{cont})) \ \textsf{else} \ \delta \\ & + \ \tau \cdot bp'(\text{cont} \cdot \langle c_0 \rangle) \end{aligned}$$

As will be shown in section 2.8.3, it can be proved that $bp'(\langle\rangle)$ is equivalent to a process $\tau \cdot g$ defined as

$$g ::= \text{get}!c_0 \cdot g + \Omega$$

The alternative Ω captures the fact that the process may perform infinite internal computation; i.e., the process can perform an infinite number of no longer observable put actions.

Synchronization The communication mechanism introduced so far suffices to describe communication between two processes. But Ada provides a mechanism whereby a collection of tasks terminate only if all agree to do so. This is, in effect, a form of multiway synchronization (no values are communicated).

The new action $\alpha\sharp\cdot$ is a multiway synchronization action. The term $\alpha\sharp \cdot p$ denotes a process that can synchronize via the label α and that will then behave like p. All processes that are parallel composed must synchronize. For example

$$\begin{aligned} & \alpha\sharp \cdot p_1 \mid \alpha\sharp \cdot p_2 \mid \ldots \mid \alpha\sharp \cdot p_n \\ = \ & \alpha\sharp \cdot (p_1 \mid p_2) \mid \alpha\sharp \cdot p_3 \mid \ldots \mid \alpha\sharp \cdot p_n \\ = \ & \ldots \\ = \ & \alpha\sharp \cdot (p_1 \mid p_2 \mid \ldots \mid p_n) \end{aligned}$$

Note that in $\alpha\sharp \cdot p \mid \beta!c_0 \cdot \alpha\sharp \cdot q$ the first process does not have the option to perform $\alpha\sharp\cdot$ without the cooperation of the second process.

Clearly, one may want to limit the synchronization requirement to a subset of all process in a system. This can be done using the hiding operator. Hiding is defined such that

$$\partial\{\alpha\}.\alpha\sharp \cdot p = \partial\{\alpha\}.p$$

Quantification Consider again the buffer process defined earlier

$$\begin{aligned} b(\text{cont}) ::= \;& \text{put}?x \cdot b(\text{cont} \cdot \langle x \rangle) \\ & + \text{ if } \neg\text{empty}(\text{cont}) \text{ then } \text{get}!\text{fst}(\text{cont}) \cdot b(\text{rest}(\text{cont})) \text{ else } \delta \end{aligned}$$

A program that uses a buffer normally creates a new instance $b(\langle\rangle)$ as in the example $p \mid b(\langle\rangle)$. Suppose, that the programmer failed to initialize the buffer content to the empty sequence. Then the new buffer instance behaves like $b(c)$ for some random content c. More precisely, the behavior is a choice $b(c_1) + b(c_2) + \ldots$ for all possible content values c_i. The new process term

$$\Sigma_i p$$

captures this idea. $\Sigma_i p$ is the possibly infinite choice $p[v_1/i] + p[v_2/i] + \ldots$ for all possible values of i.

Concatenation Finally, given two processes p_1 and p_2, the term $p_1; p_2$ denotes the sequential composition of the two processes, i.e. it consists of the behavior of p_1 followed by the behavior of p_2. The need for sequential composition stems from procedure calls. Procedural abstraction may be used to encapsulate a sequence of actions. Abstractly, this can be described by a process. Thus, it is natural to view procedures as processes and the sequential composition of procedures as sequential composition of the respective processes.

On the other hand, action prefixing is important because it provides a binding and scoping mechanism. For example, the process

$$(\alpha?x \cdot p); q$$

is different from

$$\alpha?x \cdot (p; q)$$

since in the former the binding of x is limited to p while in the latter x will be bound in $p; q$.

Equality It is obvious that different terms in process logic may denote the "same" process. Consider the process

$$a \cdot (b \cdot \epsilon + b \cdot \epsilon)$$

which performs an a and a b action and will then stop. There is no way to distinguish this behavior from that of the process

$$a \cdot b \cdot \epsilon$$

Thus, these two processes should be regarded as observationally equivalent. Two processes should be considered equal if they are observationally equivalent. How can such and equality be defined?

The equality used in process logic is based on bisimulation ([101, 87]). The idea here is that two processes are equivalent ($\equiv_B$) if, in some intuitive sense, each can simulate the behavior of the other. Bisimulation is formally defined in section 2.6.

Approximation The purpose of process logic is, of course, to use it for specifying tasking programs. Suppose, that s is a process specification and p is a process that describes the actual behavior of an agent or program (for now we ignore the question of how p may be determined). We must define precisely what it means for p to satisfy the specification s.

One possibility would be to say that a process p satisfies a specification s if p and s are the same process, i.e., $p = s$. Even though this is a reasonable definition it is not adopted here. Rather, the desired relation should provide some form of *approximation* of fixed points through loop invariants.

In section 2.8.2 the relation $\Rightarrow$ will be formally defined. For processes s and p relation $s \Rightarrow p$ holds if p is bisimulation equivalent to s whenever p terminates. Thus $\Rightarrow$ results in partial correctness specifications since $s \Rightarrow p$ is vacuously true for non-terminating p.

Trivial Specifications One of the problems with the definition of approximation is that it is not possible to leave the behavior of a process unspecified. More formally, there is no process p such that $p \Rightarrow q$ for arbitrary q. To address this problem it is necessary to introduce a new *don't care* process $\top$ with the property: $\top \Rightarrow q$ for any q. To see the utility of this construct consider a program guaranteed to start in an initial state that satisfies A. In order to show that the program behaves like process s it suffices to specify **if** A **then** s **else** $\top$; i.e., if A does not hold the program may behave arbitrarily.

1.4 Ada Verification

The previous section explained on some intuitive level how processes and their behavior can be described in process logic. It is shown here how this logic can be used to prove the correctness of tasking programs.

One of the important features of process logic is that predicate transformers very similar to the ones for sequential programs can be used to determine the process associated with a task. This section gives an informal description of the principles of using predicate transformers for describing the semantics of tasking programs. A more detailed definition of Ada tasking is presented in section 3.2.

1.4.1 Principles

The key idea is that a program will be annotated with terms of process logic. A process term placed at a point in the program asserts that the program, if started at this point, will behave as described by this process. The process associated with a concurrent program can then be computed by a suitable "concurrent predicate transformer" Wc. Consider first the sequential predicate transformer $Wp[\![\mathtt{S}]\!]A$ that gives the weakest liberal precondition ([50]) of program S under postcondition A. Precondition $B = Wp[\![\mathtt{S}]\!]A$ can be interpreted as follows: If the program is started in some initial state in which B holds and if the program terminates it will satisfy its postcondition (A).

In process logic the precondition of S can be expressed as

$$\mathsf{if}\ B\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top$$

This says that, if started in a state in which B holds and the program terminates, then the program terminates (in a state its postcondition A holds) and otherwise it may behave arbitrarily. This precondition depends on the postcondition A in the sense that the postcondition determines what should be considered a successful termination (ϵ). Note that this is a partial correctness statement: the precondition makes no guarantee of termination.

In general, the process term $\mathsf{if}\ A\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top$ corresponds to the sequential assertion A. Thus, for sequential program S the concurrent predicate transformer specializes to

$$Wc[\![\mathtt{S}]\!](\mathsf{if}\ A\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top) = \mathsf{if}(\mathsf{then}\ Wc\,\mathsf{else}[\![\mathtt{S}]\!]A)B\epsilon\top.$$

Now consider a sequential program that inputs a character and then performs the same computation as S above. Traditionally, such a program might be described by modeling the input stream and by treating this stream as a global variable. In process logic, the behavior of the program can be described by the process

$$\mathrm{get}?c \cdot \mathsf{if}\ B\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top$$

which says that if the program reads a character c from event *get* and if then B is true, then it may terminate in the desired final state. Note that $\mathrm{get}?c \cdot p$ creates a binding for c and the predicate B my depend on the value of c. Thus preconditions can be viewed as a condition on the environment in which the program is to be executed as well as the initial state: the environment must send a character c via get and the predicate B must hold in the initial state. Preconditions that are processes are referred to as *pre-processes*. Similarly, a postcondition that is a process is referred to as a *post-process*.

As a concrete example, consider the program

```
declare
    integer x;
begin
```

```
        get(x);
        if x <= 1 then
            put ("error");
        elsif prime (x) then
            put ("yes");
        else
            put ("no");
        end if;
    end;
```

For simplicity assume that the semantics of put and get are defined by channels put and get. The obvious pre-process for post-process ϵ is

$$\begin{array}{l} \text{get}?x \cdot \ \text{if}(x <= 0)\,\text{then} \\ \qquad\quad \text{put!“error”} \cdot \epsilon \\ \qquad \text{else} \\ \qquad\quad \text{if prime}(x)\,\text{then} \\ \qquad\qquad \text{put!“yes”} \cdot \epsilon \\ \qquad\quad \text{else put!“no”} \cdot \epsilon \end{array}$$

Note that the trivial post-process ϵ is appropriate. The behavior of the program is completely captured in the pre-process and the final state becomes irrelevant. Also note that this process will deadlock if it is not executed in an environment that will accept the *put* actions.

1.4.2 Restrictions

This section defines the subset of Ada tasking that can be covered by our approach as well as the kinds of properties that can be proved. Some of the restrictions limitations are in principle nature while others have been stipulated to reduce the complexity of the problem.

1.4.2.1 Termination In principle, termination is expressible in process logic since $\Omega \neq \epsilon$. This particular application of the logic, however, results in a partial correctness framework. The problem is inherent in the use of loop invariant: an invariant only proves that a certain property is maintained, not that progress is made. As a result, it is not possible to prove termination properties for concurrent programs. More generally, it will be possible to prove *safety* properties but not *liveness* properties.

In principle, process logic would not require invariants since weakest pre-processes of loops are expressible. For instance, the program

```
while x /= 0 loop x := x - 1; end loop;
```

with post-process ϵ has the pre-process $p(x)$ where

$$p(y) ::= \text{if } y \neq 0 \text{ then } p(y-1) \text{ else } \epsilon$$

which can be shown to be equal to (see section 2.8.3)

$$\mathsf{if}\, x \geq 0\, \mathsf{then}\, \epsilon\, \mathsf{else}\, \Omega$$

This means that the program will behave like ϵ if $x \geq 0$ and will diverge otherwise.

When using invariants to reason about this loop, it is possible to pick the invariant ϵ. This leads to the verification condition

$$\epsilon \Rightarrow \mathsf{if}\, x \neq 0\, \mathsf{then}\, \epsilon[x - 1/x]\, \mathsf{else}\, \epsilon$$

or $\epsilon \Rightarrow \epsilon$ and the pre-process ϵ. This is a reasonable statement in a partial correctness setting and it is sound since the pre-process ϵ is stronger than the weakest pre-process $\mathsf{if}\, x \geq 0\, \mathsf{then}\, \epsilon\, \mathsf{else}\, \Omega$, i.e.

$$\epsilon \Rightarrow \mathsf{if}\, x \geq 0\, \mathsf{then}\, \epsilon\, \mathsf{else}\, \Omega$$

1.4.2.2 Timing As an obvious consequence of the above restriction it will not be possible to reason about time. Reasoning about the timing behavior of concurrent programs is even more complex than reasoning about termination. An adequate description of timing requires assumptions about the speed of execution of an Ada program and needs to consider program optimizations, processor speeds, operating system scheduling, and other factors. The present approach treats delay statements as if they had zero delay. Similar treatment applies to selective waits and timed entry calls.

1.4.2.3 Priorities In order to deal with priorities, the formal model needs to be extended. There has been some work (e.g. [32] and [63]) on adding priorities to process algebra. But in addition, the definition of priorities requires that entry queues be modeled explicitly in the semantics (see 3.2.4). For instance, problems such as priority inversion must be captured by any semantics that deals with priorities.

1.4.2.4 Task Status The given semantics of tasks is abstract in the sense that it does not model certain implementation notions. Concepts such as the status of a task (e.g. whether the task is active or terminated) as well as explicit entry queues are not modeled by our semantics. As a result, it is not possible to reason about the following attributes:

T'CALLABLE
T'COUNT
T'TERMINATED

1.4.2.5 Abort statement The current form of process logic cannot describe abort statements properly. Aborting a task is a form of communication that affects two processes but does not require the cooperation of both. Such a mechanism is currently not provided (see 3.2.9).

1.4.2.6 Entry queues The semantics does not model Ada's entry queues. Consequently, the 'LENGTH attribute cannot be handled. Similarly, the semantics of a conditional entry call is simple a nondeterministic choice between the call and the else clause.

Omitting entry queues form the model affords a significant simplification of the definition and of program proofs. This omission is not a serious limitation since in the absence of priorities the effect of entry queues is not observable.

1.4.2.7 Global variables Global variables can always be modeled as new processes that allow only two communications: reading and writing. This is a trivial syntactic transformation and it is assumed that the abstract program representation has been transformed to represent access to shared variables by explicit read and write operations. A semantics of these operations is given in process logic.

1.5 Related Work

1.5.1 Process Algebra

The specification language described here is based on process algebra. Process algebra has its roots in Milner's CCS ([87]) and Hoare's CSP ([57]). Later work includes ATP ([53]) and ACP ([15, 18, 19]). The relevance of ATP is that it avoids the use of internal actions and uses two different choice operators instead. Bergstra and Klop [18] provide a confluent rewrite system for ACP. Some of the axiomatization in our approach is based on their ideas.

Process algebra is a very active field of research. Current work in the area ranges from theoretical foundations to practical applications.

Theoretical work in the area is concerned with model construction and extending the algebra to value passing ([59]). Also, there have been attempts to define higher-order process algebras (e.g. [117, 118]) where processes are themselves values that can be passed as arguments. Even though Ada provides task types, and thus first-class tasks, this generality was not necessary to deal with Ada semantics.

Process algebra has the advantage over other approaches to concurrency semantics that it provides process abstractions with elegant mathematical properties. This will greatly simplify the task of specifying concurrent systems and reasoning about their properties. As is pointed out in [69] process algebra is suitable for implementation through transformation and can be used for prototyping by constructing interpreters.

The disadvantage of process algebra is that it cannot describe fairness and it is cumbersome to define shared memory. While it is possible to augment process algebra with priorities (see [32]) we chose to avoid this complication.

Properties of processes can be characterized by modal formulas (Hennessy-Milner logic [55]).

Here events of the process correspond to modal operators. Thus is is possible to state that certain events *always* or *sometimes* follow other events.

There are a number of applications of process algebra. The following are just a few. The programming language *Occam* ([61]) provides built-in concurrency primitives that direct modeled after CSP. The language is actively used in production work. The formal specification languages *LOTOS* ([22]) and *RSL* ([44]) are based on process algebra (see section 1.5.3). Process algebra has been used to specify and prove correct systolic algorithms (e.g., [54]) and protocols (e.g., [120], [73]). Additional application can be found in [14].

Another approach to defining the semantics of Ada tasking is taken by Astesiano and Reggio based on SMoLCS ([12]). SMoLCS takes a two step approach. First, a denotational style definition maps programs into an intermediate language that contains concurrency primitives. The concurrency is defined in terms of a parameterized labeled transition system that has its roots in CCS ([87]) and SCCS ([89]). Properties of the transition relation are defined algebraically.

1.5.2 Other Approaches

1.5.2.1 Modal logic First-order logic can be enriched with modal operators such as "always", written $\Box$, and "eventually", written $\Diamond$. Examples of sentences in modal logic are $\Box p$ (proposition p is always true) and $\Diamond p$ (proposition p will be eventually true). Appropriate deduction rules and axioms can be defined for modal logic. For example, $(\Box p) \Rightarrow (\Diamond p)$ holds, as does $(\Box p) \Leftrightarrow \neg(\Diamond \neg p)$.

Modal logic[2] has a number of application in semantics and verification. The typical way to use modal logic to describe concurrent programs uses an enriched program state that contains the current locus of control. It is thus possible to write *control predicates* that state that a particular statement is about to be executed. For example, the predicate $@l$ holds when control is "now" at label l. Using control predicates one can write modal statements like

$$(@l_1 \wedge p) \Rightarrow \Diamond(@l_2 \wedge q)$$

This says that if control is at label l_1 and if proposition p is true, then eventually control will reach label l_2 and proposition q will be true. See [72] for an example of the use of modal logic to reason about concurrent programs.

One difficulty with modal reasoning and the used of control predicates is that the technique is not compositional. This means, that one cannot specify and reason about components separately and later draw conclusions about their composition. As a result, the technique does not support abstraction. Also, the use of control predicates is appropriate for reasoning about sequences of computation states. Process algebra abstracts from the state and considers the sequences of observable actions (that cause state changes). Hennessy-Milner logic

[2] In the context of concurrency many authors use the term *temporal logic*

([55]) is a modal logic for reasoning about processes. In essence, a process is viewed as an abstract program that executes observable actions.

Statements of a programming language can themselves be viewed as modal operators. The sentence $\boxed{\texttt{S}}p$ means "after execution of S proposition p holds". This is to say that $\boxed{\texttt{S}}p$ is the precondition of S for post-process p. This view is discussed extensively in the literature on *dynamic logic* ([52]).

1.5.2.2 Axiomatic Methods The first approaches to formally deal with concurrency were developed as extensions to axiomatic proof techniques for sequential programs. Lamport's version of concurrent Hoare logic ([71]) uses control predicates. Other approaches are closer to the original Hoare method and consider conditions under which parallel composition of program parts is well defined.

One such methods of reasoning about concurrent programs was developed by Owicki, Gries ([100]). Their method is an extension of Hoare logic based on the concept of mutual non-interference. Given two statements S1 and S2 that are executed in parallel, then the rule

$$\frac{\{P_1\}\ \texttt{S1}\ \{Q_1\}, \{P_2\}\ \texttt{S2}\ \{Q_2\}}{\{P_1 \wedge P_2\}\ \texttt{S1} \parallel \texttt{S2}\ \{Q_1 \wedge Q_2\}}$$

is valid if S1 and S2 do not *interfere*. The notion of interference can be defined more precisely in terms of variables read and written by concurrent program parts (see [100]).

Similar to the non-interference test, de Roever introduced a proof technique based on the "cooperation test" ([46]). This technique is based on conventional annotation with input output conditions. For a communication action e, the proof of an individual process may assume a particular behavior, e.g. $\{p\}e\{q\}$. Based on such assumptions the partial correctness of a single process is established. In order to combine several processes, it is necessary to show that the individual proofs cooperate. This means that if $\{p\}e\{q\}$ was assumed in the proof of one process, then the proofs of all other processes must be based on the same assumptions about e. The basic method is refined by introducing a global invariant. Using this invariant it is possible to reason about values being communicated and properties of the combined program. The global invariant may be assumed in the proof of individual program parts but it must also be shown that all program parts preserve the invariant. Consequently, the individual proofs depend on the global invariant and the technique is not compositional.

In [45] Gerth defines the semantics of the Ada rendezvous using the cooperation test.

1.5.2.3 History Sequences Various researchers have used history sequences to given an axiomatic semantics of concurrency. The core idea of this approach can be summarized as follows.

Communication is described by a history of communication events. Within one process a local, unshared history variable h is assumed. The semantics of passing a message to another

process is described by appending a send action to h. Receiving a message is modeled by appending a receive action h.

Given statements S1 and S2 that communicate only by message passing, we have the following proof rule

$$\frac{\{P_1 \wedge h = \langle\rangle\}\ \texttt{S1}\ \{Q_1 \wedge h = H{\downarrow}s_1\}, \{P_2 \wedge h = \langle\rangle\}\ \texttt{S2}\ \{Q_2 \wedge h = H{\downarrow}s_2\}}{\{P_1 \wedge P_2\}\ \texttt{S1} \parallel \texttt{S2}\ \{Q_1 \wedge Q_2\}}$$

Here H is the assumed global history of both processes. $H{\downarrow}s_1$ is the projection of the global history H onto the event sequence as seen by S1. The rule can be read as follows: If there is a global history H such that the messages sent by S1 are the message received by S2 and vice versa, then the two statements and be executed in parallel with the combined effect on the state. A description of this technique can be found in [116] and [80].

1.5.3 Other Specification Languages

Process algebra is the basis for the development of other concurrency specification languages. LOTOS and RSL are two important examples since both are concerned with specifications of real systems and go beyond the realm of academic exercises.

1.5.3.1 LOTOS LOTOS (Language of Temporal Ordering Specification) is a specification language based on process algebra. It was designed specifically to specify distributed systems and protocols. Standardization of the language by ISO is underway.

LOTOS is very similar to process logic. The following are some of the similarities between the two systems:

1. The syntax and semantics of value passing are virtually identical. The main difference is that in LOTOS binding occurrences of variables are explicitly typed.
2. The conditional of process logic corresponds to boolean guards in LOTOS.
3. The infinite sum (Σ) corresponds to the LOTOS *choice* operator.
4. LOTOS *synchronization gates* provide a way to express synchronization between multiple processes similar to the synchronization events in process logic. The key difference is that LOTOS specifies synchronization gates as part of a special parallel composition operator while process logic uses a special kind of event to indicate synchronization.

One important feature of LOTOS, not present in process logic, is the *disabling operator*. In effect $p[> q$ is a process that behaves like p until q makes a transition. From this point on only q transitions are possible. Transitions of p are disabled. This construct is very useful in defining the semantics of the Ada abort statement. A similar construct is not included in process logic since its formal basis is not fully understood.

In addition to process specifications LOTOS includes a sublanguage for the specification of abstract data types (ACT ONE [41]). In contrast, process logic only assumes some first-order language. The details of this language are left unspecified. In the case of Penelope, of course, the underlying first-order language is the Larch shared language.

1.5.3.2 RSL The RAISE project (Rigorous Approach to Industrial Software Engineering) was part of the European ESPRIT effort and involved participants from industry and academia. The team developed the RAISE specification language (RSL) for formally specifying software systems ([44]). RSL employs CLEAR-style algebraic definitions of abstract data types and uses CSP/CCS-style specifications of concurrency. In RSL processes are typed according to the channels (events) through which they can communicate. The concurrency constructs are very similar to those used in process logic. The main semantic differences are the use of internal and external choice operators (similar to ATP [53]) and the use of **stop** to denote both normal termination and deadlock. Using both internal and external choice operators avoids the need for internal actions. Internal actions are mathematically more difficult to describe. But they are closer to the user's computational model and may therefore be easier to understand. The inability of RSL to distinguish deadlock from normal termination is a serious problem.

The following is a RSL specification of a buffer process.

$$
\begin{aligned}
\text{buffer}(b) \equiv\ & \text{empty?}; \text{buffer}(\langle\rangle) \\
& [] \\
& \textsf{let } v = \text{add?} \textsf{ in } \text{buffer}(b \& \langle\rangle) \textsf{ end} \\
& [] \\
& \textsf{if } b \neq \langle\rangle \textsf{ then } \text{get!}(\text{first}(b)); \text{buffer}(\text{rest}(b)) \textsf{ else } \textbf{stop} \textsf{ end}
\end{aligned}
$$

The corresponding specification in process logic differs only in the syntax:

$$
\begin{aligned}
\text{buffer}(b) := \ & \text{empty?} \cdot \text{buffer}(\langle\rangle) && (1) \\
& + \text{add?}v \cdot \text{buffer}(b \& \langle\rangle) && (2) \\
& + \textsf{ if } b \neq \langle\rangle \textsf{ then } \text{get!first}(b) \cdot \textsf{ else } \delta && (3)
\end{aligned}
$$

1.5.4 Support Tools

Several research projects are in the stage of tool development including analysis tools and special purpose theorem provers. It can be expected that some of these tools will become usable and available in the near term. There are two classes of theorem provers, those based on a model-checking approach and those based on an inference system. Working systems of the former class are *AUTO* ([24]), Squiggle ([23]), and Workbench ([33]). One version of the workbench has extended to include priorities ([63]). Examples of the second class are CRLAB ([97]) and PSF ([81]). A comparison of these and other tools can be found in [60].

Another tool for reasoning about process algebra is PAM (Process Algebra Manipulator, [77]). PAM implements a general rewrite engine and mechanisms for various forms of inductions. The system can be configured for different process algebra formalisms by defining suitable equations (rewrite rules).

Process algebra can be expressed in HOL ([48]) and the HOL prover has been used to show bisimulation equivalence of process terms ([96]).

1.6 Organization

The following section contains a rigorous description of process logic including its syntax, semantics, and proof theory. Section 3 describes the application of process logic to specify the semantics of Ada and ways to specify and verify concurrent programs. Only the principles are discussed here and this section does not constitute a complete formal definition of Ada tasking semantics.

In section 4 process logic is applied in several examples. This illustrates techniques for the systematic development of tasking programs, strategies for developing annotations, and proof techniques. The examples point out areas where the basic mechanisms need to be extended by more abstract specification concepts in order to become practical.

2 Formal Basis

2.1 Notation

Lambda notation has its usual meaning, e.g. $\lambda x.fxx$. Type information is omitted where it can be inferred. Sometimes function application is written as juxtaposition, e.g. fx. Functions may be curried. Functions can be redefined point-wise as

$$f[d \to e] = \lambda x.\,\mathsf{if}\, x = d\,\mathsf{then}\, e\,\mathsf{else}\, f\ x$$

Substitution of x for y in t is written as $p[x/y]$. The notation for parallel substitution is $p[x_1/y_1, \ldots, x_n/y_n]$.

2.2 Domains

Definition 1 *A domain is a set D with a partial order $\sqsubseteq$ such that*

1. *$\bot \in D$ and $\bot \sqsubseteq d$ for all $d \in D$.*
2. *Every $\sqsubseteq$ chain has a least upper bound.*
3. *There is a countable base $D_B \subseteq D$ such that every $d \in D$ is the least upper bound of some D_B chain.*

The importance of domains is that equations involving + (sum), × (product), and → (continuous function) constructors have solutions in domains. Further, continuous functions on domains have unique least fixed points. A more detailed discussion of domains can be found in [51].

Describing concurrent and non-deterministic programs requires powerdomains. I.e., a constructor $\mathbf{P}$ is needed that constructs the domain of subsets of a given domain. It is then possible to solve domain equation involving $\mathbf{P}$ (see e.g. [51, 78, 79]).

One powerdomain construction due to Symth ([113]) defines $\mathbf{P}_S(D)$ is defined as

$$\mathbf{P}_S(D) = \{S \subset D | S \text{ is non-empty and finite or } \bot \in S\}$$

An ordering (Egli-Milner) $\sqsubseteq$ on $\mathbf{P}_S(D)$ is defined as follows

$$S \sqsubseteq T ::= \begin{cases} \forall s \in S. \exists t \in T. s \sqsubseteq t \\ \forall t \in T. \exists s \in S. s \sqsubseteq t \end{cases}$$

One problem with this construction is that $\mathbf{P}_S(D)$ does not contain the empty set.

The powerdomain construction proposed by Plotkin [103] does not suffer this problem and the following is based on Plotkin powerdomains. Abramsky shows in [2] how to extend the latter to a constructor $\mathbf{P}^0$ that generates a powerdomain that contains the empty set.

For the details of the $\mathbf{P}^0$ construction see [2]. Here only the following properties are of interest.

- $\mathbf{P}^0$ is a functor, i.e., it maps a function $f \in D_1 \to D_2$ to a function $\mathbf{P}^0 f \in \mathbf{P}^0(D_1) \to \mathbf{P}^0(D_2)$.
- Further, the following operations are defined:

$\{\!\{\}\!\} \in \mathbf{P}^0(D)$	Empty set
$\{\!\{.\}\!\} \in D \to \mathbf{P}^0(D)$	Singleton set
$\uplus \in \mathbf{P}^0(D) \to \mathbf{P}^0(D) \to \mathbf{P}^0(D)$	Union
$\biguplus \in \mathbf{P}^0(\mathbf{P}^0(D)) \to \mathbf{P}^0(D)$	Join

Using the construction given in [2] it can be shown that $(\mathbf{P}^0, \{\!\{.\}\!\}, \biguplus)$ is a monad as defined in [121]. Following Wadler ([121]) $\mathbf{P}^0$ is a "monad with zero" ($\lambda x.\{\!\{\}\!\}$) and therefore admits comprehension with filters. Thus, terms such as

$$\{\!\{fx \mid x \in X, px\}\!\} \in \mathbf{P}^0(D_2)$$

for $X \in \mathbf{P}^0(D_1)$ and continuous $f \in D_1 \to D_2$ and predicate p has meaning as defined in [121]:

$$\{\!\{t \mid x \in X\}\!\} \;=\; \mathbf{P}^0(\lambda x.t)X$$

$$\begin{aligned}
\{\!|t \mid x \in X, \ldots|\!\} &= \biguplus\{\!|\{\!|t \mid \ldots|\!\} \mid x \in X|\!\} \\
\{\!|t \mid p|\!\} &= \mathsf{if}\, p\, \mathsf{then}\, \{\!|t|\!\}\, \mathsf{else}\, \{\!|\,|\!\} \\
\{\!|t \mid p, \ldots|\!\} &= \biguplus\{\!|\{\!|t \mid \ldots|\!\} \mid p|\!\} \\
&= \mathsf{if}\, p\, \mathsf{then}\, \{\!|t \mid \ldots|\!\}\, \mathsf{else}\, \{\!|\,|\!\}
\end{aligned}$$

For example, the term $\{\!|fx \mid x \in X, px|\!\}$ expands as follows

$$\begin{aligned}
\{\!|fx \mid x \in X, px|\!\} &= \biguplus\{\!|\{\!|t \mid px|\!\} \mid x \in X|\!\} \\
&= \biguplus\{\!|\,\mathsf{if}\, px\, \mathsf{then}\, \{\!|t|\!\}\, \mathsf{else}\, \{\!|\,|\!\} \mid x \in X|\!\} \\
&= \biguplus(\mathbf{P}^0(\lambda x.\, \mathsf{if}\, px\, \mathsf{then}\, \{\!|t|\!\}\, \mathsf{else}\, \{\!|\,|\!\})X)
\end{aligned}$$

2.3 First-order Logic

Terms of process logic will contain terms and formulas of some first-order language $\mathcal{L}$ defined as follows[3]:

The syntax is defined as follows:

$A, B \in \mathit{Pred}$	Predicate Symbols
$f, g \in \mathit{Func}$	Function Symbols
$c \in \mathit{Const}$	Constants
$x, v \in \mathit{Var}$	Variables
$t \in \mathit{Term}$	Terms
$A \in \mathit{Form}$	Formulas

where

$$\begin{aligned}
t &::= v \mid c \mid f(t_1, \ldots, t_n) \\
A &::= t_1 = t_2 \mid P(t_1, \ldots, t_n) \mid \mathsf{false} \mid A_1 \to A_2 \mid \forall x.A \mid \ldots
\end{aligned}$$

The usual set of boolean connectives and existential quantifiers will be used; they can be defined from the above in the obvious way.

An $\mathcal{L}$-Structure $\mathcal{N} = (D, I)$ consists of a domain D and an interpretation I such that

1. $I(P) \subseteq D^n$ for n-ary predicate P
2. $I(f) \in D^n \to D$ for n-ary function f

[3]Very little changes if a different logic is substituted in the definition of process logic

3. $I(c) \in D$ for every constant.

A $\mathcal{N}$-valuation $\nu \in V = Var \rightarrow D$ assigns a value in D to every variable of $\mathcal{L}$. Every $\nu \in V$ extends to all terms in the obvious way, we write $\nu(t)$ to denote the value of term t under valuation ν.

We write

$$\mathcal{N} \models_\nu A$$

if the formula A is true under the valuation ν. $\mathcal{N} \models A$ says that A is true in $\mathcal{N}$ under all valuations. Equivalently, the notation $\mathcal{N}[\![A]\!]\nu$ is used to denote the truth of $\mathcal{N} \models_\nu A$.

2.4 Processes

For (first-order) language $\mathcal{L}$ the language of processes $\mathcal{P}(\mathcal{L})$ is defined as follows[4].

$\alpha, \beta \in Lab$	Labels (or channels)
$H \subset Lab$	Subset of labels
$p, q \in Proc$	Processes
$P, Q \in Pfun$	Process functions
$f, g \in Pvar$	Function symbols

A and t refer to formulas and terms of $\mathcal{L}$ respectively.

$$\begin{aligned} P &::= \lambda x.p \mid f \mid \mu f.P \\ p &::= \delta \mid \epsilon \mid \Omega \mid \tau \cdot p \mid \text{if } A \text{ then } p \text{ else } q \mid \partial H.p \mid \\ &\quad\ \alpha\sharp \cdot p \mid \alpha!t \cdot p \mid \alpha?x \cdot p \mid p + q \mid p \mid q \mid p;q \mid \Sigma_{x \in V} p \end{aligned}$$

Processes distinguish deadlock (δ) and normal termination (ϵ). The treatment is similar to ACP [15] in that $p + \delta = p$ holds for all p. This differs from the treatment in [5] where $p + \epsilon = p$ holds instead.

ACP uses concatenation (";") as the basic process constructors and CCS based systems use action prefixing ("$\cdot$"). Process logic contains both constructs. Action prefixing is appropriate for receive actions since they are binding operations: the process following a receive action is the scope of the received value. On the other hand, process concatenation turns out to needed for the semantics of a procedure calls.

[4]The syntax has been changed drastically from earlier versions. The main reason is to be consistent with other work in the area. The original notation was heavily influenced by a modal logic approach. The present notation has its roots in ACP, CCS, ECS.

The introduction of sequential composition leads to *context free* processes, i.e., a language that (in the absence of value passing) is strictly more powerfull than one that only uses action prefixing. For instance,

$$p ::= (a \cdot p; b \cdot \epsilon) + \epsilon$$

is a process that generates the context free language of actions $a^n b^n$ which cannot be defined with action prefixing alone since this defines only regular languages. Note, however, that by introducing auxiliary state variables we can define

$$\begin{aligned} p &::= p_1(0) \\ p_1(n) &::= a \cdot p_1(n+1) + p_2(n) \\ p_2(n) &::= \mathsf{if}\, n = 0\, \mathsf{then}\, \epsilon\, \mathsf{else}\, b \cdot p_2(n-1) \end{aligned}$$

The notation $\Sigma_{x \in X} p$ is used to denote the (possibly infinite) choice over all values $x_i \in X$, $p[x_1/x] + p[x_2/x] + \ldots$ We write $\Sigma_x p$ if the domain of x is clear from the context.

Constructs for sending and receiving values were originally proposed by Milne and Milner in [83]. More recently, a variation was proposed in [59].

Because of the presence of value passing and the conditional construct, the behavior of a process depends on the state. And the definition of recursive processes needs to account for changing state. For this reason the syntax contains general process functions, i.e., functions from values to processes and a least fixed point construct μ for such functions. The notational shorthand $f(x) ::= p$ is used to make recursive processes more readable. The term $f(t)$ means $(\mu f.\lambda x.p)(t)$ in the context of a definition $f(x) ::= p$. Special care is needed in manipulating formulas that used this abbreviated notation. For instance $f(t)[e/y] = f(t)$ is generally *false* even if y is not free in t since it may be free in the definitional equation of f.

2.5 Operational Semantics

An operational semantics of process logic can be given as a labeled transition system. Let $\mathcal{A}$ be the set of actions defined as follows:

τ represents an internal computation,

$\alpha!v$ represents the sending of value v via event α,

$\alpha?v$ represents the reception of value v via event α, and

$\alpha\sharp$ represents the synchronization with event α.

Function $lab \in \mathcal{A} \rightarrow Lab \cup \{\tau\}$ determines the label of an action:

$$\begin{aligned} lab(\tau) &= \tau \\ lab(\alpha!v) &= \alpha \end{aligned}$$

$$\begin{aligned} lab(\alpha?v) &= \alpha \\ lab(\alpha\sharp) &= \alpha \end{aligned}$$

For $a \in \mathcal{A}$ we write $a \cdot p$ for the process that performs a and then behaves like p.

A labeled transition system is a relation $\rightsquigarrow \subset Proc \times \mathcal{A} \times Proc$. If a triple $\langle p, a, q \rangle \in \rightsquigarrow$ we write $p \overset{a}{\rightsquigarrow} q$. By convention $a \in \mathcal{A}$ is any action. $p \overset{a}{\rightsquigarrow} q$ means that the process p can perform the action a and transition to process q.

Note that the possible transitions for process p depend on the state (i.e., the values of the free variables of p). Thus the following definitions assume an arbitrary but fixed $\mathcal{L}$-structure $\mathcal{N}$ and $\mathcal{N}$-valuation ν. More pedantically, one might write $\mathcal{N} \models_\nu p \overset{a}{\rightsquigarrow} q$, meaning that in a state ν process p can perform a and then behave like q.

$$\alpha!t \cdot p \overset{\alpha!t}{\rightsquigarrow} p \qquad \alpha?x \cdot p \overset{\alpha?t}{\rightsquigarrow} p[t/x]$$

$$\tau \cdot p \overset{\tau}{\rightsquigarrow} p \qquad \alpha\sharp \cdot p \overset{\alpha\sharp}{\rightsquigarrow} p$$

$$\frac{p \overset{a}{\rightsquigarrow} q}{p + r \overset{a}{\rightsquigarrow} q} \qquad \frac{p \overset{a}{\rightsquigarrow} q}{r + p \overset{a}{\rightsquigarrow} q}$$

$$\frac{p[c/x] \overset{a}{\rightsquigarrow} q}{\Sigma_x p \overset{a}{\rightsquigarrow} q}$$

$$\frac{p \overset{a}{\rightsquigarrow} q,\ lab(a) \notin H}{\partial H.p \overset{a}{\rightsquigarrow} \partial H.q} \qquad \frac{p \overset{\alpha\sharp}{\rightsquigarrow} q,\ \partial H.q \overset{a}{\rightsquigarrow} r,\ \alpha \in H}{\partial H.p \overset{a}{\rightsquigarrow} r}$$

$$\frac{p \overset{a}{\rightsquigarrow} q,\ B}{\textsf{if } B \textsf{ then } p \textsf{ else } r \overset{a}{\rightsquigarrow} q} \qquad \frac{p \overset{a}{\rightsquigarrow} q,\ \neg B}{\textsf{if } B \textsf{ then } r \textsf{ else } p \overset{a}{\rightsquigarrow} q}$$

Note that B refers to the truth of B is the given valuation ν.

$$\frac{p \overset{a}{\rightsquigarrow} q,\ a \neq \alpha\sharp}{(r \mid p) \overset{a}{\rightsquigarrow} (r \mid q)} \qquad \frac{p \overset{a}{\rightsquigarrow} q,\ a \neq \alpha\sharp}{(p \mid r) \overset{a}{\rightsquigarrow} (q \mid r)}$$

$$\frac{p \overset{\alpha?t}{\rightsquigarrow} q,\ p' \overset{\alpha!t}{\rightsquigarrow} q'}{p \mid p' \overset{\tau}{\rightsquigarrow} q \mid q'} \qquad \frac{p \overset{\alpha\sharp}{\rightsquigarrow} p',\ q \overset{\alpha\sharp}{\rightsquigarrow} q'}{(p \mid p') \overset{\alpha\sharp}{\rightsquigarrow} (q \mid q')}$$

$$\frac{p \overset{a}{\rightsquigarrow} q}{p;r \overset{a}{\rightsquigarrow} q;r} \qquad \frac{p \overset{a}{\rightsquigarrow} q}{\epsilon;p \overset{a}{\rightsquigarrow} q}$$

$$\frac{P[\mu f.P/f](t) \overset{a}{\rightsquigarrow} q}{\mu f.P(t) \overset{a}{\rightsquigarrow} q} \qquad \frac{p[t/x] \overset{a}{\rightsquigarrow} q}{(\lambda x.p)(t) \overset{a}{\rightsquigarrow} q}$$

The relation $\rightsquigarrow$ can be extended to sequences of actions in the obvious way. In particular, we write

$$p \overset{a\dagger}{\rightsquigarrow} q \text{ iff } p \overset{\tau}{\rightsquigarrow} \ldots \overset{\tau}{\rightsquigarrow} p' \overset{a}{\rightsquigarrow} q' \overset{\tau}{\rightsquigarrow} \ldots \overset{\tau}{\rightsquigarrow} q$$

i.e., if there are zero or more internal transitions and an a transition that transform p into q. Similarly,

$$p \overset{a*}{\rightsquigarrow} q \text{ iff } \begin{cases} p \overset{a\dagger}{\rightsquigarrow} q & \text{when } a \neq \tau \\ p \overset{a\dagger}{\rightsquigarrow} q \textbf{ or } p = q & \text{when } a = \tau \end{cases}$$

includes the reflexive case $p \overset{\tau*}{\rightsquigarrow} p$.

The unary relation $\surd \subset Proc$ defines terminated processes. A process is in $\surd$ if it cannot perform any further actions but is not deadlocked. Instead of $p \in \surd$ we write $p\surd$. Again, the following definitions assume some fixed valuation ν. $\surd$ is the smallest relation such that

$$\begin{array}{rll} \epsilon\surd & & \\ p\surd & \text{implies} & (\partial H.p)\surd \\ p\surd, q\surd & \text{implies} & (p+q)\surd, (p \mid q)\surd, (p;q)\surd \\ \forall c.(p[c/x]\surd) & \text{implies} & (\Sigma_x p)\surd \\ p\surd, B & \text{implies} & (\textsf{if } B \textsf{ then } p \textsf{ else } q)\surd \\ p\surd, \neg B & \text{implies} & (\textsf{if } B \textsf{ then } q \textsf{ else } p)\surd \\ p[t/x]\surd & \text{implies} & (\lambda x.p)(t)\surd \\ P[\mu f.P/f](t)\surd & \text{implies} & (\mu f.P)(t)\surd \end{array}$$

Similarly, $\downarrow \subset Proc$ is the relation defining *definite* processes, i.e., those processes that do not diverge without performing any action. Again, $p \in \downarrow$ is written $p \downarrow$ and $p \downarrow$ means that $p \downarrow$ for all valuations in which A is true. $\downarrow$ is the smallest relation such that

$$\begin{array}{rlll} \epsilon \downarrow & & & \\ \delta \downarrow & & & \\ a \cdot p \downarrow & & & \\ p \downarrow & \text{implies} & (\partial H.p) \downarrow & \\ p \downarrow & \text{implies} & (p;q) \downarrow & \\ p\surd, q \downarrow & \text{implies} & (p;q) \downarrow & \\ p \downarrow, q \downarrow & \text{implies} & (p+q) \downarrow \text{ and } (p \mid q) \downarrow & \end{array}$$

$$\begin{aligned} \forall c.(p[c/x]\downarrow) &\text{ implies } (\Sigma_x p)\downarrow \\ p\downarrow, B &\text{ implies } (\textsf{if } B \textsf{ then } p \textsf{ else } q)\downarrow \\ p\downarrow, \neg B &\text{ implies } (\textsf{if } B \textsf{ then } q \textsf{ else } p)\downarrow \\ p[t/x]\downarrow &\text{ implies } (\lambda x.p)(t)\downarrow \\ P[\mu f.P/f](t)\downarrow &\text{ implies } (\mu f.P)(t)\downarrow \end{aligned}$$

Next, $\Downarrow \subset Proc$ is the set of definite processes that cannot perform an infinite sequence of τ actions. Formally $\Downarrow$ is the smallest relations such that

$$p \Downarrow \text{ if } \left(p\downarrow \wedge (\forall q.p \overset{\tau}{\rightsquigarrow} q \rightarrow q \Downarrow)\right)$$

Finally, $\nabla \subset Proc$ is the set of processes that will not deadlock before performing a non-τ action, i.e. $p\nabla$ if any state q reachable through τ transitions from which not further τ transitions are possible will be a terminated state, formally

$$p\nabla \text{ iff } \left(\forall q.p \overset{\tau^*}{\rightsquigarrow} q \wedge \neg\exists q'.q \overset{\tau}{\rightsquigarrow} q' \rightarrow q\surd\right)$$

2.6 Process Equivalence

Obviously, two different process terms can denote two processes that are not observably different. The key question is what me mean by "observable". There are a number of different equivalence relations proposed in the literature. *Bisimulation* ([101]) appears to be the most natural definition. The definition says that two processes are equivalent if any action that can be performed by one can be performed by the other such that the resulting new processes are equivalent. The literature distinguishes between *weak* and *strong* bisimulation. The difference is that weak bisimulation assumes that the performance of an internal action is not observable.

One strong argument in favor of bisimulation is the *modal characterization theorem.* The theorem says that two processes are bisimulation equivalent if and only if they satisfy the same sets of formulas in Hennessy-Milner logic.

A different process equivalence, called *branching bisimulation*, has been proposed ([47]). Branching bisimulation is a smaller relation than bisimulation, i.e., there are processes that are distinguished by branching bisimulation but that are weakly bisimilar. Branching bisimulation provides a reasonable definition of process equivalence since, as for weak bisimulation, there is a modal characterization theorem: Two processes are branching bisimilar if and only if they satisfy the same set of formulas in *Hennessy-Milner logic with until* (see [37]).

In absence of strong evidence that other equivalence relations lead to simpler proof procedures we choose weak bisimulation for process logic.

This section first defines a bisimulation preorder $\sqsubseteq_B$. Intuitively, two processes p and q are in the relation $p \sqsubseteq_B q$ if either p diverges or if p and q have the same behavior. A process

equivalence can then be defined as $\equiv = \sqsubseteq_B \cap \sqsubseteq_B^{-1}$. It turns out that $\equiv$ is not a congruence. Next, a slightly smaller relation $\sqsubseteq_C$ is defined such that $\sqsubseteq_C \cap \sqsubseteq_C^{-1}$ will be a congruence. The latter relation will be process equality.

The bisimulation preorder $p \sqsubseteq_B q$ is the largest relation such that $p \sqsubseteq_B q$ if and only if

1. if $p \overset{a}{\rightsquigarrow} p'$ there exists a q' such that $q \overset{a*}{\rightsquigarrow} q'$ and $p' \sqsubseteq_B q'$,
2. if $p \Downarrow$ then
   (a) if $q \overset{a}{\rightsquigarrow} q'$ there exists a p' such that $p \overset{a*}{\rightsquigarrow} p'$ and $p' \sqsubseteq_B q'$,
   (b) $q \Downarrow$
   (c) $p\nabla$ iff $q\nabla$

The definition of bisimulation preorder is slightly different from the traditional one (e.g. [101]). The difference is the distinction between processes that terminate normally from those that deadlock which is due to [5].

Bisimulation preorder expresses a very intuitive ordering between processes: if $p \sqsubseteq_B q$, then q is better that p in the sense that

- if p can perform a sequence of observable actions, then q can perform the same sequence of observable actions
- if p converges, then
  - q converges
  - q cannot perform actions that cannot be performed by p, and
  - if p can deadlock then so can q.

It is easy to check that all process terms except $p+q$ are monotonic with respect to $\sqsubseteq_B$. The following example shows that $p+q$ is not monotonic. It is immediate that $\tau \cdot \alpha! \cdot p \sqsubseteq_B \alpha! \cdot p$.

$$\tau \cdot \alpha! \cdot p + \beta! \cdot q \quad \not\sqsubseteq_B \quad \alpha! \cdot p + \beta! \cdot q \tag{4}$$

since $(\tau \cdot \alpha! \cdot p + \beta! \cdot q) \overset{\tau}{\rightsquigarrow} \alpha! \cdot p$ is a possible behavior for which there is no corresponding transition in $\alpha! \cdot p + \beta! \cdot q$.

Two processes p and q are equivalent if $p \sqsubseteq_B q$ and $q \sqsubseteq_B p$, i.e., $\equiv$ is defined as $\sqsubseteq_B \cap \sqsubseteq_B^{-1}$. Unfortunately, $\equiv$ is not a congruence. This follows immediately from the fact that $\sqsubseteq_B$ is not monotonic for $+$: Let $p \equiv q$, then $p \sqsubseteq_B q$. The previous example (equation 1 above) shows that there exists a context C such that $C[p] \not\sqsubseteq_B C[q]$ and consequently $C[p] \not\equiv C[q]$.

In general, if $\leq$ is any preorder on a term algebra and if all operator symbols are monotonic with respect to $\leq$, then $\leq \cap \leq^{-1}$ is a congruence. This property can be used to construct a congruence for processes as follows. Let $p \sqsubseteq_C q$ be the largest relation such that $p \sqsubseteq_C q$ if and only if

1. if $p \overset{a}{\rightsquigarrow} p'$ there exists a q' such that $q \overset{a+}{\rightsquigarrow} q'$ and $p' \sqsubseteq_B q'$,
2. if $p \Downarrow$ then
   (a) if $q \overset{a}{\rightsquigarrow} q'$ there exists a p' such that $p \overset{a+}{\rightsquigarrow} p'$ and $p' \sqsubseteq_B q'$,
   (b) $q \Downarrow$
   (c) $p\nabla$ iff $q\nabla$

Note that this definitions differs only from that of $\sqsubseteq_B$ in clauses (1) and (2a) where $\overset{a*}{\rightsquigarrow}$ has been replaced by $\overset{a+}{\rightsquigarrow}$. Also observe that this change only applies to the top level; states reachable need only be related by $\sqsubseteq_B$.

All process terms are monotonic with respect to $\sqsubseteq_C$. Thus $\sqsubseteq_C \cap \sqsubseteq_C^{-1}$ is a congruence and we define this relation to be process equality.

Milner ([88]) proposes an alternate, equivalent definition of equality

$$p = q \text{ iff } \forall r.(p + r) \equiv (q + r)$$

The preorder $\sqsubseteq_C$ is also the relation used to approximate fixed points. Since $\sqsubseteq_C$ is monotonic, the following rule rule is sound:

$$\frac{P[I/f](x) \sqsubseteq_C I}{\mu f.R(t) \sqsubseteq_C I[t/x]}$$

In other words, given the fixed point $\mu f.P$ and an invariant I for which one can show the *verification condition* $P[I/f](x) \sqsubseteq_C I$ then this invariant is an approximation to the fixed point.

2.7 The "don't care" Process

There is one slight problem with the definitions given so far: it is not possible to write specifications that leave the behavior of a process unspecified. To correct this problem a new process $\top$ is introduced. The *don't care* process $\top$ is defined such that $p \sqsubseteq_C \top$ for all processes p. All process terms other than the conditional are strict with respect to $\top$.

Using $\top$ it is possible to write specifications like **if** A **then** p **else** $\top$ that specifies a process that behaves like p if A holds and is unspecified otherwise.

Finally, the notation $p \Rightarrow q$ means $q \sqsubseteq_C p$, suggesting the correspondence to implication in the sequential case. In fact, the formula

$$\textbf{if } A \textbf{ then } p \textbf{ else } \top \Rightarrow \textbf{if } B \textbf{ then } p \textbf{ else } \top$$

is true if and only if $A \rightarrow B$.

2.8 Proof Theory

2.8.1 Equations

The following equations hold for process terms.

Internal computations

$$\begin{aligned} a \cdot \tau \cdot p &= a \cdot p && \text{(N1)} \\ \tau \cdot p + p &= \tau \cdot p && \text{(N2)} \\ a \cdot (p + \tau \cdot q) &= a \cdot (p + \tau \cdot q) + a \cdot q && \text{(N3)} \end{aligned}$$

See [91] for a discussion of the equality rules. In particular, the seemingly natural axiom $a \cdot (p + \tau \cdot q) = a \cdot (p + q) + a \cdot q$ is not sound.

Hiding

$$\begin{aligned} \partial H.\delta &= \delta && \text{(H1)} \\ \partial H.\epsilon &= \epsilon && \text{(H2)} \\ \partial H.\Omega &= \Omega && \text{(H3)} \\ \partial H.\tau \cdot p &= \tau \cdot \partial H.p && \text{(H4)} \\ \partial H.\,\mathsf{if}\, A \,\mathsf{then}\, p \,\mathsf{else}\, p &= \mathsf{if}\, A \,\mathsf{then}\, \partial H.p \,\mathsf{else}\, \partial H.q && \text{(H5)} \\ \partial H.\alpha\sharp \cdot p &= \alpha \in H : \partial H.p, \alpha\sharp \cdot \partial H.p && \text{(H6)} \\ \partial H.\alpha!t \cdot p &= \alpha \in H : \delta, \alpha!t \cdot \partial H.p && \text{(H7)} \\ \partial H.\alpha?x \cdot p &= \alpha \in H : \delta, \alpha?x \cdot \partial H.p && \text{(H8)} \\ \partial H.(p_1 + p_2) &= \partial H.p_1 + \partial H.p_2 && \text{(H9)} \\ \partial H.(p_1; p_2) &= \partial H.p_1; \partial H.p_2 && \text{(H10)} \end{aligned}$$

Conditionals

$$\begin{aligned} \mathsf{if\,false\,then}\, p \,\mathsf{else}\, q &= q && \text{(G1)} \\ \mathsf{if\,true\,then}\, p \,\mathsf{else}\, q &= p && \text{(G2)} \end{aligned}$$

Alternatives

$$\begin{aligned} p + (q + r) &= (p + q) + r && \text{(A1)} \\ p + q &= q + p && \text{(A2)} \\ \delta + p &= p && \text{(A3)} \\ p + p &= p && \text{(A4)} \end{aligned}$$

Concatenation

$$\begin{aligned}
\delta; p &= \delta && \text{(J1)}\\
\epsilon; p &= p && \text{(J2)}\\
\Omega; p &= \Omega && \text{(J3)}\\
(\text{if } A \text{ then } p \text{ else } q); r &= \text{if } A \text{ then}(p; r) \text{ else}(q; r) && \text{(J4)}\\
(\alpha\sharp \cdot p); q &= \alpha\sharp \cdot (p; q) && \text{(J5)}\\
(\alpha!t \cdot p); q &= \alpha!t \cdot (p; q) && \text{(J6)}\\
(\alpha?x \cdot p); q &= (\alpha?x \cdot (p; q)) \text{ where } x \text{ not free in } q && \text{(J7)}\\
(p_1 + p_2); q &= (p_1; q) + (p_2; q) && \text{(J8)}\\
(p_1; p_2); p_3 &= p_1; (p_2; p_3) && \text{(J9)}
\end{aligned}$$

Recursion

$$(\mu f.P)t = P[\mu f.P/f](t) \qquad \text{(R1)}$$

Concurrent composition

An equational definition of concurrent composition can be given in term of two auxiliary terms (following [18]):

$p \between q$ is like $p \mid q$ except that the first action that is a communication of p with the environment.

$p \mid_c q$ is a process that performs the internal action of communicating between p and q and that then behaves as the concurrent composition of the resulting processes.

With these definitions concurrent composition can be defined as

$$\begin{aligned}
p_1 \mid (p_2 \mid p_3) &= (p_1 \mid p_2) \mid p_3 && \text{(P1)}\\
p \mid q &= (p \between q) + (q \between p) + (p \mid_c q) && \text{(P2)}
\end{aligned}$$

Given $\alpha \neq \beta$ then

$$
\begin{array}{rcll}
p \mid_c q & = & q \mid_c p & \text{(C1)} \\
(p \mid_c q) \mid_c r & = & p \mid_c (q \mid_c r) & \text{(C2)} \\
\delta \mid_c p & = & \delta & \text{(C3)} \\
\epsilon \mid_c p & = & \delta & \text{(C4)} \\
\Omega \mid_c p & = & \Omega & \text{(C5)} \\
\tau \cdot p \mid_c q & = & \delta & \text{(C6)} \\
(\textsf{if } A \textsf{ then } p \textsf{ else } p') \mid_c q & = & \textsf{if } A \textsf{ then} (p \mid_c q) \textsf{ else} (p' \mid_c q) & \text{(C7)} \\
\alpha\sharp \cdot p \mid_c \beta\sharp \cdot q & = & \textsf{if } \alpha = \beta \textsf{ then } \alpha\sharp \cdot (p \mid q) \textsf{ else } \delta & \text{(C8)} \\
\alpha!t \cdot p \mid_c \alpha?x \cdot q & = & \textsf{if } \alpha = \beta \textsf{ then } \tau \cdot (p \mid q[x/t]) \textsf{ else } \delta & \text{(C9)} \\
\alpha!t \cdot p \mid_c \beta\sharp \cdot q & = & \delta & \text{(C10)} \\
\alpha!t \cdot p \mid_c \beta!t' \cdot q & = & \delta & \text{(C11)} \\
\alpha?y \cdot p \mid_c \beta?x \cdot q & = & \delta & \text{(C12)} \\
\alpha?y \cdot p \mid_c \beta\sharp \cdot q & = & \delta & \text{(C13)} \\
(p + q) \mid_c r & = & p \mid_c r + q \mid_c r & \text{(C14)}
\end{array}
$$

$$
\begin{array}{rcll}
\delta \between p & = & \delta & \text{(S1)} \\
\epsilon \between p & = & \delta & \text{(S2)} \\
\Omega \between p & = & \Omega & \text{(S3)} \\
\tau \cdot p \between q & = & \tau \cdot (p \mid q) & \text{(S4)} \\
(\textsf{if } A \textsf{ then } p \textsf{ else } p') \between q & = & \textsf{if } A \textsf{ then} (p \between q) \textsf{ else} (p' \between q) & \text{(S5)} \\
\alpha\sharp \cdot p \between q & = & \delta & \text{(S6)} \\
\alpha!t \cdot p \between q & = & \alpha!t \cdot (p \mid q) & \text{(S7)} \\
\alpha?x \cdot p \between q & = & \alpha?x \cdot (p \mid q) \text{ where } x \text{ not free in } q & \text{(S8)} \\
(p + q) \between r & = & p \between r + q \between r & \text{(S9)}
\end{array}
$$

Where y in (S8) is a new variable.

Consequences

The following properties can be derived from the above axioms:

$$
\begin{array}{rcll}
p_1 \mid p_2 & = & p_1 \mid p_2 & \text{(P3)} \\
\alpha\sharp \cdot p \mid \alpha\sharp \cdot q & = & \alpha\sharp \cdot (p \mid q) & \text{(P4)}
\end{array}
$$

P3: Using equation C1

$$
\begin{array}{rcl}
p_1 \mid p_2 & = & (p_1 \between p_2) + (p_2 \between p_1) + (p_1 \mid_c p_2) \\
& = & (p_2 \between p_1) + (p_1 \between p_2) + (p_2 \mid_c p_1) \\
& = & p_1 \mid p_2
\end{array}
$$

P4:

$$
\begin{array}{rcl}
\alpha\sharp \cdot p \mid \alpha\sharp \cdot q & = & \alpha\sharp \cdot p \between \alpha\sharp \cdot q + \alpha\sharp \cdot q \between \alpha\sharp \cdot p + \alpha\sharp \cdot p \mid_c \alpha\sharp \cdot q \\
& = & \delta + \delta + \alpha\sharp \cdot (p \mid q) \\
& = & \alpha\sharp \cdot (p \mid q)
\end{array}
$$

2.8.2 Approximation

$$\frac{}{p \Rightarrow p} \qquad \frac{p \Rightarrow q,\ q \Rightarrow r}{p \Rightarrow r}$$

$$\frac{}{\top \Rightarrow q} \qquad \frac{}{p \Rightarrow \Omega}$$

$$\frac{p \Rightarrow q}{C[p] \Rightarrow C[q]}$$

where $C[\ldots]$ is an arbitrary context.

$$\frac{P(x) \Rightarrow Q(x)}{\mu f.P(t) \Rightarrow \mu f.Q(t)}$$

$$\frac{I(x) \Rightarrow P[I/f](x)}{I(t) \Rightarrow \mu f.P(t)}$$

More generally, if R is an admissible predicate, the following fixed point induction rule applies to process terms.

$$\frac{R(\Omega), \forall x.R(f(x)) \rightarrow \forall x.R(P(x))}{R(\mu f.P(t))}$$

Where the predicates $\lambda p.(p \sqsubseteq_C p_c)$, $\lambda p.(p \sqsupseteq_C p_c)$, and $\lambda p.(p = p_c)$ are all admissible for arbitrary processes p_c.

2.8.3 Example Proofs

The following proof shows the equality of $bp'(\langle\rangle) = \tau \cdot g$ given in section 1.3. Recall that

$$g := get!c_0 \cdot g + \tau \cdot g$$

and

$$bp'(\text{cont}) ::= \text{if } \neg\text{empty(cont) then get!fst(cont)} \cdot bp'(\text{rest(cont)}) \text{ else } \delta + \tau \cdot bp'(\text{cont} \cdot \langle c_0 \rangle)$$

Let c_0^n stand for the sequence of length n of values c_0. We show the slightly stronger theorem

$$bp'(c_0^n) = g \text{ for all } n > 0.$$

The original claim then follows since by definition

$$bp'(\langle\rangle) = bp'(c_0^0) = \tau \cdot bp'(c_0^1) = \tau \cdot g$$

The proof of $bp'(c_0^n) = g$ for $n > 1$ proceeds in two steps showing $g \Rightarrow bp'(c_0^n)$ and $bp'(c_0^n) \Rightarrow g$ respectively.

For the first case the following instance of the induction rule is used:

$$\frac{\begin{array}{c}\forall n > 0.(g \Rightarrow \Omega) \\ (\forall n > 0.(g \Rightarrow bp'(c_0^n))) \rightarrow \\ (\forall n > 0.g \Rightarrow \mathsf{if}\ \neg\mathrm{empty}(c_0^n)\ \mathsf{then}\ \mathrm{get!fst}(c_0^n) \cdot bp'(\mathrm{rest}(c_0^n))\ \mathsf{else}\ \delta \\ + \tau \cdot bp'(c_0^n \cdot \langle c_0 \rangle))\end{array}}{\forall n > 0.(g \Rightarrow bp'(c_0^n))}$$

To show the second premise of the rule consider two cases, $n > 1$ and $n = 1$. For $n > 1$

$$\begin{array}{rcl} & = & g \\ & = & \mathrm{get!}c_0 \cdot g + \tau \cdot g \\ & \Rightarrow & \mathrm{get!}c_0 \cdot bp'(c_0^{n-1}) + \tau \cdot bp'(c_0^{n+1}) \\ & = & \mathsf{if}\ \neg\mathrm{empty}(c_0^n)\ \mathsf{then}\ \mathrm{get!}c_0 \cdot bp'(c_0^{n-1})\ \mathsf{else}\ \delta + \tau \cdot bp'(c_0^{n+1}) \end{array}$$

For $n = 1$

$$\begin{array}{rcl} g & = & \mathrm{get!}c_0 \cdot g + \tau \cdot g \\ & \Rightarrow & \mathrm{get!}c_0 \cdot bp'(c_0^1) + \tau \cdot bp'(c_0^2) \\ & = & \mathrm{get!}c_0 \cdot \tau \cdot bp'(c_0^1) + \tau \cdot bp'(c_0^2) \\ & = & \mathrm{get!}c_0 \cdot (\delta + \tau \cdot bp'(c_0^1)) + \tau \cdot bp'(c_0^2) \\ & = & \mathrm{get!}c_0 \cdot bp'(c_0^0) + \tau \cdot bp'(c_0^2) \\ & = & \mathsf{if}\ \neg\mathrm{empty}(c_0^n)\ \mathsf{then}\ \mathrm{get!}c_0 \cdot bp'(c_0^{n-1})\ \mathsf{else}\ \delta + \tau \cdot bp'(c_0^{n+1}) \end{array}$$

In the other direction we use fixed point induction on the definition of g. The rule instance is

$$\frac{\begin{array}{c}\forall n > 0.(bp'(c_0^n) \Rightarrow \Omega) \\ (\forall n > 0.(bp'(c_0^n) \Rightarrow g)) \rightarrow (\forall n > 0.bp'(c_0^n) \Rightarrow get!c_0 \cdot g + \tau \cdot g)\end{array}}{\forall n > 0.(bp'(c_0^n) \Rightarrow g)}$$

Again, the second premise is shown by case analysis, considering $n > 1$ and $n = 1$.

As second example consider the process $p(x) ::= \mathsf{if}\ x \neq 0\ \mathsf{then}\ p(x-1)\ \mathsf{else}\ \epsilon$ given in section 1.4.2. We prove that $\mathsf{if}\ x \geq 0\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \Omega \Rightarrow p(x)$. The fixed point rule require to show the premise

$$\begin{array}{l} \forall x.(\mathsf{if}\ x \geq 0\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \Omega \Rightarrow p(x)) \\ \rightarrow \\ \forall x.(\mathsf{if}\ x \geq 0\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \Omega \Rightarrow \mathsf{if}\ x \neq 0\ \mathsf{then}\ p(x-1)\ \mathsf{else}\ \epsilon) \end{array}$$

which, by monotonicity, becomes

$$\forall x.(\text{if } x \neq 0 \text{ then if } x - 1 \geq 0 \text{ then } \epsilon \text{ else } \Omega \text{ else } \epsilon \Rightarrow \text{if } x \geq 0 \text{ then } \epsilon \text{ else } \Omega)$$

The latter breaks into three cases:

$x = 0$

$$\begin{aligned} \text{if } x \geq 0 \text{ then } \epsilon \text{ else } \Omega &= \epsilon \\ &\Rightarrow \epsilon \\ &= \text{if } x \neq 0 \text{ then if } x - 1 \geq 0 \text{ then } \epsilon \text{ else } \Omega \text{ else } \epsilon \end{aligned}$$

$x > 0$

$$\begin{aligned} \text{if } x \geq 0 \text{ then } \epsilon \text{ else } \Omega &= \epsilon \\ &\Rightarrow \epsilon \\ &= \text{if } x - 1 \geq 0 \text{ then } \epsilon \text{ else } \Omega \\ &= \text{if } x \neq 0 \text{ then if } x - 1 \geq 0 \text{ then } \epsilon \text{ else } \Omega \text{ else } \epsilon \end{aligned}$$

$x < 0$

$$\begin{aligned} \text{if } x \geq 0 \text{ then } \epsilon \text{ else } \Omega &= \Omega \\ &\Rightarrow \Omega \\ &= \text{if } x - 1 \geq 0 \text{ then } \epsilon \text{ else } \Omega \\ &= \text{if } x \neq 0 \text{ then if } x - 1 \geq 0 \text{ then } \epsilon \text{ else } \Omega \text{ else } \epsilon \end{aligned}$$

To show equality of if $x \geq 0$ then ϵ else Ω and $p(x)$ consider the cases $x < 0$ and $x \geq 0$. Using the above approximation the first case gives

$$\text{if } x \geq 0 \text{ then } \epsilon \text{ else } \Omega = \Omega \Rightarrow p(x)$$

which implies $p(x) = \Omega$. In the second case ($x \geq 0$) a simple induction argument shows that $p(x) = \epsilon$.

2.9 Denotational Semantics

This sections gives a denotational semantics of process terms. The semantics of process terms is already well defined by the operational semantics given in terms of labeled transition systems. The denotational semantics is useful to establish the connection between process terms and programming language semantics. The semantics of a concurrent language can be defined in terms of power domains. A predicate transformer semantics for the same language will map programs into process terms. Showing the soundness of such a predicate

transformer definition amounts to proving that the denotation of the process term derived for a program is the same as the denotation of the program itself (see section 3.1.1).

The denotational semantics should be fully abstract, which means that every property that holds in the model should also hold for the processes. More precisely, given processes p and q and their denotations $[\![p]\!]$ and $[\![q]\!]$, then one wants $p \sqsubseteq_C q$ iff $[\![p]\!] \sqsubseteq [\![q]\!]$.

In [83] Milne and Milner give a suitable domain definition for processes as

$$D = \mathbf{P}_S(\Sigma_{\mu \in Lab}(U_\mu \times (V_\mu \to D)))$$

In this construction an event μ is associated with sending a value ($\in U_\mu$) and receiving a value ($\in V_\mu$). This can be specialized to directional communication by choosing either U_μ or V_μ as the single point domain. This construction is not adequate since it does not deal with internal (τ) actions.

In [2] Abramsky give a similar construction using $\mathbf{P}^0$ which correctly models the deadlocked process but omits value passing. This domain is fully abstract for strong bisimulation but not for weak bisimulation. In [59] a similar construction is given that includes value passing.

The following is a domain suitable for process logic.

$\alpha, \beta \in Lab$			Labels
$d \in D$			Data
A_S	$=$	$Lab \times D \times P$	Send action
A_R	$=$	$Lab \times (D \to P)$	Receive action
A_X	$=$	$Lab \times P$	Synchronization action
A_I	$=$	$\{\tau\} \times P$	Internal action
$\pi \in P$	$=$	$\mathbf{P}^0(A_S + A_R + A_X + A_I + \{\epsilon\})$	Processes

The following functions on P are introduced:

$pfx \in (P \to P) \to P \to P$	$next \in P \to P$
$send \in Lab \to D \to P \to P$	$sync \in Lab \to P \to P$
$recv \in Lab \to (D \to P) \to P$	$cat \in P \to P \to P$
$comm \in P \to P \to P$	$step \in P \to P$
$hide \in P \to 2^{Lab} \to P$	$merge \in P \to P \to P$

With the definitions:

$$\begin{aligned}
pfx\, f\, \pi &= \biguplus\{\!|\, pfx\, f\, \pi_i \mid \langle \tau, \pi_i \rangle \in \pi \,|\!\} \uplus \\
&\quad \biguplus\{\!|\, f\pi \mid \emptyset \neq \{\!|\, a \mid a \in \pi, a \neq \langle \tau, \pi' \rangle \,|\!\} \,|\!\} \\
next &= pfx(\lambda\pi.\{\!|\langle \tau, \pi \rangle|\!\}) \\
send\, \alpha\, d &= pfx(\lambda\pi'.\{\!|\langle \alpha, d, \pi' \rangle|\!\}) \\
sync\, \alpha &= pfx(\lambda\pi'.\{\!|\langle \alpha, \pi' \rangle|\!\}) \\
recv\, \alpha &= \{\!|\langle \alpha, \lambda d.\, next(fd) \rangle|\!\}
\end{aligned}$$

The purpose of the definition of *pfx* is to guarantee a canonical representation as follows:

> Whenever $\langle \alpha, d, \pi' \rangle \in \pi$ then π' contains at least one choice that does not involve an internal action, i.e., $\exists a \in \pi'.a \neq \langle \tau, \pi'' \rangle$. Furthermore, if $\langle \tau, \pi'' \rangle \in \pi'$ then $\langle \alpha, d, \pi'' \rangle \in \pi$.

The same condition applies to choices of the form $\langle \alpha, \pi' \rangle \in \pi$ and $\langle \tau, \pi' \rangle \in \pi$.

$$
\begin{aligned}
merge(\pi_1, \pi_2) &= step(\pi_1, \pi_2) \uplus step(\pi_2, \pi_1) \uplus comm(\pi_1, \pi_2) \\
step(\pi_1, \pi_2) &= \biguplus \{\!\{ send\,\beta d(merge(\pi, \pi_2)) \mid \langle \beta d \pi \rangle \in \pi_1 \}\!\} \uplus \\
&\quad \biguplus \{\!\{ recv\,\beta(\lambda d'.\, merge(f d', \pi_2)) \mid \langle \beta, f \rangle \in \pi_1 \}\!\} \uplus \\
&\quad \biguplus \{\!\{ next(merge(\pi, \pi_2)) \mid \langle \tau, \pi \rangle \in \pi_1 \}\!\} \uplus \\
&\quad \{\!\{ \epsilon \mid \epsilon \in \pi_1 \}\!\} \\
comm(\pi_1, \pi_2) &= \biguplus \{\!\{ next(merge(f d, \pi)) \mid \langle \beta, d, \pi \rangle \in \pi_1, \langle \beta, f \rangle \in \pi_2 \}\!\} \uplus \\
&\quad \biguplus \{\!\{ next(merge(f d, \pi)) \mid \langle \beta, d, \pi \rangle \in \pi_2, \langle \beta, f \rangle \in \pi_1 \}\!\} \uplus \\
&\quad \biguplus \{\!\{ sync\,\beta\, merge(\pi_1', \pi_2') \mid \langle \beta, \pi_1' \rangle \in \pi_1 \langle \beta, \pi_2' \rangle \in \pi_2 \}\!\}
\end{aligned}
$$

$$
\begin{aligned}
cat(\pi_1, \pi_2) &= \biguplus \{\!\{ send\,\beta d\, cat(\pi', \pi_2) \mid \langle \beta, d, \pi' \rangle \in \pi_1 \}\!\} \uplus \\
&\quad \biguplus \{\!\{ recv\,\beta(\lambda d.\, cat(f d, \pi_2)) \mid \langle \beta, f \rangle \in \pi_1 \}\!\} \uplus \\
&\quad \biguplus \{\!\{ sync\,\alpha\, cat(\pi', \pi_2)) \mid \langle \alpha, \pi' \rangle \in \pi_1 \}\!\} \uplus \\
&\quad \biguplus \{\!\{ \tau \cdot (cat(\pi, \pi_2)) \mid \langle \tau, \pi \rangle \in \pi_1 \}\!\} \uplus \\
&\quad \biguplus \{\!\{ \pi_2 \mid \epsilon \in \pi_1 \}\!\}
\end{aligned}
$$

$$
\begin{aligned}
hide(\pi, H) &= \biguplus \{\!\{ send\,\beta d\, hide(\pi', H) \mid \langle \beta, d, \pi' \rangle \in \pi, \beta \notin H \}\!\} \uplus \\
&\quad \biguplus \{\!\{ recv\,\beta(\lambda d.\, hide(f d, H)) \mid \langle \beta, f \rangle \in \pi, \beta \notin H \}\!\} \uplus \\
&\quad \biguplus \{\!\{ sync\,\beta\, hide(\pi', H) \mid \langle \beta, \pi' \rangle \in \pi, \beta \notin H \}\!\} \uplus \\
&\quad \biguplus \{\!\{ next(hide\,\pi' \beta) \mid \langle \beta, \pi' \rangle \in \pi, \beta \in H \}\!\} \uplus \\
&\quad \biguplus \{\!\{ next(hide\,\pi' H) \mid \langle \tau, \pi' \rangle \in \pi \}\!\} \uplus \\
&\quad \{\!\{ \epsilon \mid \epsilon \in \pi \}\!\}
\end{aligned}
$$

2.9.1 Process Semantics

Using the domain P a denotational semantics of process terms is given. To deal with recursive processes environments of the form

$$\rho \in E = Pvar \to D \to P$$

States

$$\nu \in S = Var \to D$$

map variables to their values. These are the valuations for the underlying first-order logic $\mathcal{L}$.

The denotations of process terms are defined by two functions of the form

$$\mathcal{M} \in Proc \to E \to S \to P$$
$$\mathcal{M}' \in Pfun \to E \to S \to D \to P$$

defined as

$$\begin{aligned}
\mathcal{M}[\![\delta]\!]\rho\nu &= \{\!\}\\
\mathcal{M}[\![\epsilon]\!]\rho\nu &= \{\!|\epsilon|\!\}\\
\mathcal{M}[\![\Omega]\!]\rho\nu &= \{\!|\perp|\!\}\\
\mathcal{M}[\![\tau \cdot p]\!]\rho\nu &= next(\mathcal{M}[\![p]\!]\rho\nu)\\
\mathcal{M}[\![\textsf{if } A \textsf{ then } p \textsf{ else } q]\!]\rho\nu &= \textsf{if } \mathcal{N}[\![A]\!]\nu \textsf{ then } \mathcal{M}[\![p]\!]\rho\nu \textsf{ else } \mathcal{M}[\![p]\!]\rho\nu\\
\mathcal{M}[\![\partial H.p]\!]\rho\nu &= hide(\mathcal{M}[\![p]\!]\rho\nu, H)\\
\mathcal{M}[\![\alpha\sharp \cdot p]\!]\rho\nu &= sync\,\alpha(\mathcal{M}[\![p]\!]\rho\nu)\\
\mathcal{M}[\![\alpha!t \cdot p]\!]\rho\nu &= send\,\alpha(\mathcal{N}[\![t]\!]\nu)(\mathcal{M}[\![p]\!]\rho\nu)\\
\mathcal{M}[\![\alpha?x \cdot p]\!]\rho\nu &= recv\,\alpha(\lambda d.\mathcal{M}[\![p]\!]\rho\nu[d/x])\\
\mathcal{M}[\![p + q]\!]\rho\nu &= \mathcal{M}[\![p]\!]\rho\nu \uplus \mathcal{M}[\![q]\!]\rho\nu\\
\mathcal{M}[\![p_1 \mid p_2]\!]\rho\nu &= merge(\mathcal{M}[\![p_1]\!]\rho\nu, \mathcal{M}[\![p_2]\!]\rho\nu)\\
\mathcal{M}[\![p \between q]\!]\rho\nu &= step(\mathcal{M}[\![p]\!]\rho\nu, \mathcal{M}[\![q]\!]\rho\nu)\\
\mathcal{M}[\![p \mid_c q]\!]\rho\nu &= comm(\mathcal{M}[\![p]\!]\rho\nu, \mathcal{M}[\![q]\!]\rho\nu)\\
\mathcal{M}[\![p; q]\!]\rho\nu &= cat(\mathcal{M}[\![p]\!]\rho\nu, \mathcal{M}[\![q]\!]\rho\nu)\\
\mathcal{M}[\![P(t)]\!]\rho\nu &= \mathcal{M}'[\![P]\!]\rho\nu(\mathcal{N}[\![t]\!]\nu)
\end{aligned}$$

$$\begin{aligned}
\mathcal{M}'[\![\mu f.P]\!]\rho\nu &= \textsf{fix}\,\lambda\psi \in (D \to P).\mathcal{M}'[\![P]\!]\rho[f \to \psi]\nu\\
\mathcal{M}'[\![\lambda x.p]\!]\rho\nu &= \lambda d \in D.\mathcal{M}[\![p]\!]\rho\nu[x \to d]\\
\mathcal{M}'[\![f]\!]\rho\nu &= \rho[\![f]\!]
\end{aligned}$$

Conjecture: $\mathcal{M}$ is fully abstract with respect to $\sqsubseteq_C$.

3 Semantics and Verification

3.1 Concepts

Process logic as defined in section 2 can be used to describe abstract observable behavior. This section discusses how the semantics of programming languages can be described using process transformers and how the correctness of programs can be proved in this framework. A process transformer is to concurrent programs what a predicate transformer is to sequential ones.

3.1.1 Predicate Transformers

The concept of predicate transformers was first introduced by Dijkstra in [40]. A predicate transformer for a program fragment S is a mapping from a postcondition p to a precondition q. A precondition q that is true in the state before execution of S guarantees that, if S terminates normally, p will hold in the final state. There are different possible interpretations of preconditions. A *weakest liberal precondition* is one that is implied by any other precondition that satisfies the above interpretation. Finally, Dijkstra's *weakest preconditions* also guarantee termination of the given program.

The sequential predicate transformers used in the Penelope system are formally related to a continuation semantics. This section briefly reviews this connection. Section 3.1.2 shows that process transformers stand in the same formal relationship with a continuation semantics based on power domains.

A continuation is a mapping from program states to answers, i.e., program results. The continuation associated with a point in the program describes the result when the program is started at this point (label) with a particular state. Continuation semantics describes the meaning of programs by defining the continuation *before* a statement in terms of the continuation *after* the statement, i.e., the denotation of a statement is a continuation transformer.

The relation between continuations and predicates has long been recognized (see e.g. [84]). The following view is based on [107]: an assertion at a point in the program is viewed as a description of a set of possible continuations. Assume that for a fixed program point the continuation θ describes the effect of the remainder of the computation. Let $A_0 \subset A$ be a subset of the "desirable" answers of the program, then A_0 and continuation θ define a predicate P on states $s \in S$ in the following sense:

$$P(s) \text{ iff } \theta s \in A_0.$$

I.e., P is true for a state, if the program gives a desirable result when started at the given point in this state.

Since a continuation semantics defines the continuation before a statement in terms of the continuation following the statement, it effectively gives us a predicate transformer that produces the precondition (for a desirable result) based on the postcondition of the statement.

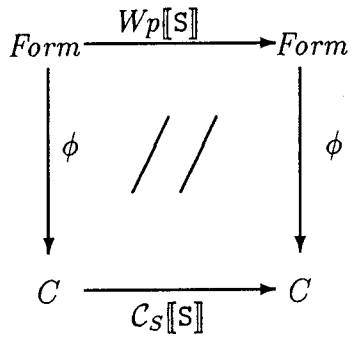

Figure 1: Soundness of Predicate Transformers

The soundness of predicate transformers is illustrated in figure 1. Assume that $\mathcal{C}_S$ is a continuation semantics that maps sequential program S into a function from continuations C to continuations. The notion of "desirable" answer can be captured by assuming that answers are truth values. It is then possible to map a first-order assertion into a continuation via $\phi(A) = \lambda s.\mathcal{N}[\![A]\!]s$. The predicate transformers are sound if the diagram in figure 1 commutes.

This transformation is formalized and extended with the notion of invariants in [107] and forms the basis of the Ada predicate transformer definition used in the Penelope system ([109]). The key idea in combining invariants with predicate transformers is to allow approximations, i.e., preconditions that are stronger than necessary. In this case user-provided invariants and procedure pre- and postconditions can be used to approximate fixed points associated with loops and recursive procedures.

This approach leads to a verification system based on the proof of first-order verification conditions. At the same time, the semantic definition can make use of all of the definitional idioms used in denotational semantics. In particular, expression continuations can be used to define expression with side-effect and environments can be used for defining goto statements, exceptions and other non-local transfer of control.

3.1.2 Application to Concurrency

Predicate transformers for concurrent program are formally related to a continuation semantics in very much the same way as in the sequential case outlined above.

In the sequential case, a continuation is a mapping from states to answers. In the concurrent case, a continuation maps states to processes. The answer of a concurrent program is its observable behavior.

A concurrent programming language (without shared variables) can be defined by a continuation semantics where continuations map states to powerdomain P as defined in section 2.9. Given a continuation θ mapping $s \in S$ to $\pi \in \mathcal{P}$, it defines a set of process terms p such

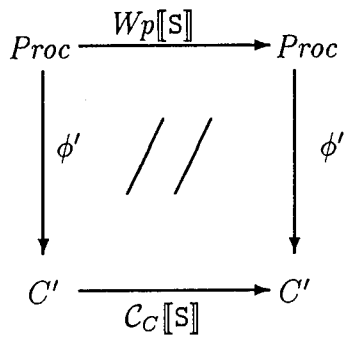

Figure 2: Soundness of Process Transformers

that

$$\mathcal{M}[\![p]\!]s \Rightarrow \theta s$$

The relation between the continuation semantics and the process transformers is illustrated in figure 2. Here $\mathcal{C}_C$ is a concurrent continuation semantics that maps program S into a function from continuations ($C' = S \rightarrow P$) to continuations. A term in process logic defines a continuation (C') via function $\phi'(p) = \lambda s.\mathcal{M}[\![p]\!] \perp s$. The process transformers are sound if the diagram 2 commutes.

3.1.3 A Special Case: Sequential Programs

Obviously, sequential programs are special cases of concurrent ones. Thus, it is reasonable to expect that using concurrent predicate transformers for a sequential program would lead to the same proof obligations as traditional predicate transformers. It is show here that this is indeed the case. The following argument shows (i) that, in the sequential case, a process transformer definition can be systematically be derived from a first-order predicate transformer definition and (ii) that everything provable in one formalism is provable in the other.

The transformation Ψ maps first-order formulas into corresponding process logic terms according to

$$\Psi(A) = \mathsf{if}\ A\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top$$

This transformation extends to annotated programs naturally, by replacing every annotation A by $\Psi(A)$, resulting in a program with process logic annotations. Now let Wp be a predicate transformer for sequential programs (S) with first-order annotations, producing first-order preconditions ($\in$ *Form*). A process transformer Wc for (sequential) programs with process logic annotations ($\in$ *Proc*) producing pre-processes can be constructed systematically from Wp as follows:

Wp constructs first-order preconditions $P \in \mathcal{L}$ from assertions A by (i) substitution $P[u/x]$, (ii) quantification $\forall x.P$, and (iii) formation of conditionals $\mathsf{if}\ A\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$, where A are

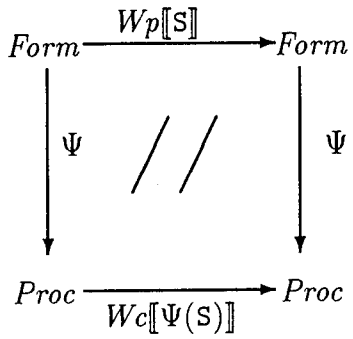

Figure 3: Relation between sequential and concurrent predicate transformers

first-order formulas. A corresponding predicate transformer Wc can be constructed by the following transformation

Wp		Wc
$\_[u/x]$	$\Rightarrow$	$\_[u/x]$
$\forall x.\_$	$\Rightarrow$	$\Sigma_x.\_$
if A then $\_$ else $\_$	$\Rightarrow$	if A then $\_$ else $\_$

Furthermore, whenever Wp generates the verification condition $A \rightarrow B$, the Wc generates the verification condition $\Psi(A) \Rightarrow \Psi(B)$.

Wc constructed from Wp by this transformation will be consistent with Wp the sense indicated in the diagram in Figure 3, i.e.,

$$Wc[\![\Psi(\mathsf{S})]\!]\Psi(A) = \Psi(\,Wp[\![\mathsf{S}]\!]A)$$

To prove that the diagram (Figure 3) commutes we show

1. $\Psi(A[u/x]) = \Psi(A)[u/x]$
2. $\Psi(\forall x.A) = \Sigma_x.\Psi(A)$
3. $\Psi(\mathsf{if}\ A\ \mathsf{then}\ A_1\ \mathsf{else}\ A_2) = \mathsf{if}\ A\ \mathsf{then}\ \Psi(A_1)\ \mathsf{else}\ \Psi(A_2)$

The proof of this is immediate:

1.

$$\begin{aligned}\Psi(A[u/x]) &= \mathsf{if}\ A[u/x]\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top \\ &= (\mathsf{if}\ A\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top)[u/x] \\ &= \Psi(A)[u/x]\end{aligned}$$

2.

$$\begin{aligned}\Psi(\forall x.A) &= \mathsf{if}\ \forall x.A\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top \\ &= \Sigma_x.(\mathsf{if}\ A\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top) \\ &= \Sigma_x.\Psi(A)\end{aligned}$$

Where the equality

$$\mathsf{if}\ \forall x.A\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top = \Sigma_x.(\mathsf{if}\ A\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top)$$

follows by case analysis. If $\forall x.A$ then $\Sigma_x.(\mathsf{if}\ A\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top) = \epsilon$. If $\neg A[x_0/x]$ for some x_0, then $\Sigma_x.(\mathsf{if}\ A\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top) = \top$ since $+$ is strict with respect to $\top$.

3.

$$\begin{aligned}\Psi(\mathsf{if}\ A\ \mathsf{then}\ A_1\ \mathsf{else}\ A_2) &= \mathsf{if}\ \mathsf{if}\ A\ \mathsf{then}\ A_1\ \mathsf{else}\ A_2\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top \\ &= \mathsf{if}\ A\ \mathsf{then}\ \mathsf{if}\ A_1\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top\ \mathsf{else} \\ &\qquad \mathsf{if}\ A_2\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top \\ &= \mathsf{if}\ A\ \mathsf{then}\ \Psi(A_1)\ \mathsf{else}\ \Psi(A_2)\end{aligned}$$

Finally, we can show that everything that is provable with first-order predicate transformers is also provable in process logic. Consider the first-order verification condition

$$A \rightarrow Wp[\![\mathsf{S}]\!]B$$

If translated as above, this becomes

$$\Psi(A) \Rightarrow Wc[\![\Psi(\mathsf{S})]\!]\Psi(B)$$

and, using figure 3

$$\Psi(A) \Rightarrow \Psi(Wp[\![\mathsf{S}]\!]B)$$

Thus, for every verification condition $A_1 \rightarrow A_2$ generated by Wp there is a corresponding verification condition $\Psi(A_1) \Rightarrow \Psi(A_2)$ generated by Wc. Verification condition $A_1 \rightarrow A_2$ is provable if and only if $\Psi(A_1) \Rightarrow \Psi(A_2)$ is.

Assume that $A_1 \rightarrow A_2$ holds. Then either $A_1 \wedge A_2$ or $\neg A_1$. In the first case, $\mathsf{if}\ A_1\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top \Rightarrow \mathsf{if}\ A_2\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top$ becomes $\epsilon \Rightarrow \epsilon$ which holds trivially. If $\neg A_1$, then $\mathsf{if}\ A_1\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top \Rightarrow \mathsf{if}\ A_2\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top$ becomes $\top \Rightarrow \mathsf{if}\ A_2\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top$ which also holds since $\top \Rightarrow p$ holds for arbitrary p. To show the converse, assume that $A_1 \rightarrow A_2$ is false. This means $A_1 \wedge \neg A_2$, in which case $\mathsf{if}\ A_1\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top \Rightarrow \mathsf{if}\ A_2\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top$ becomes $\epsilon \Rightarrow \top$ which is false.

3.2 Ada Tasking

Based on the preceding, the process transformers for the sequential part of Ada follow immediately from the predicate transformers. This section discusses the details of mapping

Ada tasking constructs to process logic. The presentation here is simplified in order to focus on the principles involved. A detailed formal predicate transformer definition for Ada tasking is beyond the scope of this document

To simplify the exposition the presentation uses abstract simplified syntax. The focus is on tasking semantics and many semantic details are omitted. These follow directly from their sequential definition. Also, the exposition ignores details of maintaining an environment. Phrases like "the appropriate ..." and "the ... associated with ..." indicate that the information needed can be computed from the environment.

The presentation omits the use of expression continuations and declaration continuations. Instead it is assumed that all expressions evaluate without side-effects. The definition can be extended to cover the proper Ada evaluation order rules using the same techniques as in the sequential case ([109])

3.2.1 Task Declarations

The semantics of a task is described by a process term. This term defines the communication behavior of the task, depending on the initial state. There is no post-process associated with a task since the final state of a task is not observable (the user can, of course, add an annotation at the end of the task body). The meaning of a task can simply be determined as the pre-process of the task body for the post-process ϵ.

The user may provide an optional pre-process (s) with a task specification. In this case the user's specification will be meaning of the task and a verification condition is generated to ensure that the implementation (p) satisfies this specification, i.e., the verification condition will be $s \Rightarrow p$.

There is a slight complication in the case of task types since different instances of the same task body may be activated. Obviously, their associated processes will differ, e.g., the entries (and the events associated with calls to these entries) are different. To address this issue the notion of a task value is introduced.

The domain of task values is an unbounded set of discrete elements. Task values will be used to uniquely identify task instances. A task template is a process parameterized by a task value. The meta-variable *self* is defined inside a task body and denotes the current task value. The process for a task instance can be constructed from the task template by substituting a new unique task value for for *self* in the template. The new task name is the value of the task. It can treated like any other first class value. As will be shown later, the events associated with the entries of a task instance can be constructed from the task type (which is known statically) and the task value. Thus, task values can be used to model the values of task variables as well as access values to tasks.

For uniformity it is assumed that all tasks are task types. A trivial syntactic transformation can transform declared tasks into this case.

The pre-process of a task body

```
task body I is
  D
begin
  S
end I;
```

is the process

$$p = Wc[\![\texttt{D; S}]\!](\gamma\sharp \cdot \epsilon)$$

This says that the execution of the task consists of the elaboration of the local declarations followed by the execution of the statements in the task body. The post process $\gamma\sharp \cdot \epsilon$ says that after the execution the task will synchronize with the master and all and then stop. Here γ is the synchronization label (see 3.2.3) unique to the master of task `I`. If the task declaration contains a specification q, then the verification condition $q \Rightarrow p$ is generated. The template associated with task `I` is $\lambda \mathsf{self}.p$. New instances are generated by applying this template to a new unique task name.

There is no dynamic semantics associated with task specifications.

3.2.2 Task Activation

The Ada language specifies that declared tasks are activated at the point of the "begin" of their declarative region and that allocated tasks are activated when the allocator is evaluated. It is assumed that for declared tasks the abstract syntax representation of tasking programs contains an explicit representation for the activation of static tasks. Appropriate "activate" statements follow the appropriate "begin". The semantics of the allocation is to create a new, unique task value, to initialize the static tasks variable with this value, to instantiate the task template, and to parallel compose the resulting process with the creating process.

For task activation we have

$$Wc[\![\text{activate } t]\!]p = p[x/t] \mid q[x/self]$$

where t is a static task, x is a new task value, and q is the task template associated with t. Similarly, for allocated tasks

$$Wc[\![\text{new } t]\!](\lambda v.p) = (\lambda v.p)(x) \mid q[x/self]$$

where $\lambda v.p$ is some expression continuation that receives the new task value.

Explicit representation of task activation poses one problem that so far has not been addressed. Suppose the program contains the declaration of an array of tasks with dynamic bounds. In this case, the number of tasks that need to be activated is not static and activation needs to be done through some form of looping construct (see 4.3). Any such loop

would require some form of invariant. It may be possible that suitable invariants could be constructed automatically, but this issue has not been studied. Thus, currently our technique is restricted to programs that contain only a fixed number of statically declared tasks in every scope. Note that there are no restrictions on the number of tasks being allocated.

3.2.3 Task Termination

There are two interesting aspects to Ada task termination. First, a master (roughly the enclosing scope) can only terminate if all dependent tasks have terminated. The master of a declared task is the scope of its declaration, the master of an allocated tasks is the scope that declared the access type. Secondly, tasks that include the terminate alternative will terminate only when all siblings (tasks depending on the same master) are willing to terminate and if the master's execution is is completed.

The semantics of both of these features is described in terms of synchronization events. Each master (scope declaring a task or task type) is assigned a unique synchronization label (γ). The last action in the master is to synchronize with γ. Similarly, the semantics of each terminate alternative in all dependent tasks is to synchronize with γ or, if there is no terminate alternative, the last action of a task is to synchronize with γ.

This definition, given the semantics of synchronization in process logic, ensures that all dependent tasks and their master terminate simultaneously. Terminate alternatives can only be taken when all tasks are willing to terminate. To bound the scope of the synchronization event the definition of activation needs to be revised as

$$Wc[\![\text{activate } t]\!]p = \partial\{\gamma\} \cup H_t.(p[x/t] \mid q[x/self])$$

where H_t is the set of all labels associated with entries of task t.

For example, consider the three processes

$$\begin{aligned} p &::= \gamma\sharp \cdot \epsilon + \alpha!x \cdot p \\ q &::= \gamma\sharp \cdot \epsilon + \alpha!y \cdot q \\ m &::= \alpha?z \cdot \alpha?z \cdot \gamma\sharp \cdot r \end{aligned}$$

Where p and q might be the processes describing two tasks that repeatedly send messages to α and that contain a terminate alternative. Process m might be the behavior of a master that contains p and q, i.e., m will receive two values from α and will then exit the current scope and will then continue as process r. Process r describes the remainder of the computation. Informally, the predicate transformer for the activation of p and q (actually, the activation of the respective tasks) is given as

$$\partial\{\alpha, \gamma\}.(p \mid q \mid m)$$

Using the equations in section 2.8.1 the latter term simplifies to

$$\begin{aligned} & \partial\{\alpha,\gamma\}.(p \mid q \mid m) \\ = \; & \partial\{\alpha,\gamma\}.(p \mid ((\gamma\sharp \cdot \epsilon + \alpha!y \cdot q) \mid (\alpha?z \cdot \alpha?z \cdot \gamma\sharp \cdot r))) \\ = \; & \ldots \\ = \; & \tau \cdot \partial\{\alpha,\gamma\}.r \end{aligned}$$

In addition to completing the execution of the task body, a task can terminate by executing a terminate alternative. The process transformer for the terminate alternative is

$$Wc[\![\texttt{terminate}]\!]p = \gamma\sharp \cdot \epsilon$$

Note that this definition is independent of p, indicating a non-local transfer of control.

3.2.4 Entry Calls

There are three actions associated with an entry call. First, the entry call starts. Some time later, the entry call terminates. The termination may be either normal or exceptional. Thus we have

- $t.e_\alpha$ – the label associated with starting an entry call to entry e of task value (instance) t. The caller will perform the action $t.e_\alpha!e \cdot p$ and the callee will perform $t.e_\alpha?x \cdot q$.
- $t.e_\omega$ – the label associated with normal termination of an entry call to entry e of task value t. In this case the callee will perform $t.e_\omega!e \cdot q$, returning the out and inout parameters to the caller that will perform the matching action $t.e_\omega?x \cdot p$.
- $t.e_\epsilon$ – the label associated with exceptional termination of an entry call. The callee will execute the send action, passing the exception value to the caller which will raise this exception in the calling program.

The Ada semantics of entry calls is very specific about the entry calls that are not immediately possible are queued on entry queues. In principle it would be possible to represent entry queues in our semantics. But this detailed description would be significantly more complex that the approach taken here. As a consequence of this simplification, it is not possible to reason about entry queues (e.g. their length attribute).

As a shorthand notation the task name *self* will be omitted. For example e_α means $self.e_\alpha$. The notation suggests visibility as in Ada, i.e., naming an entry refers to the local name. It is important understand the full expansion of this shorthand in cases like

$$(e_\alpha?x \cdot p)[t/self] = t.e_\alpha?x \cdot (p[t/self])$$

3.2.5 Entries and Accept Statements

There is no semantics associated with entry declarations. An accept statement performs the actions matching those of the an entry call as described in the previous section. Given

```
accept e (x : t; y : out t) do
   S
end
```

and post-process p, the pre-process is defined as

$$self.e_\alpha?x \cdot Wc[\![\texttt{S}]\!](self.e_\omega!y \cdot p)$$

In the typical case where S contains only assignments, this simplifies to

$$self.e_\alpha?x \cdot self.e_\omega!t \cdot p$$

for some term t that reflects the effect of the assignments. But in general S may contain arbitrary code including other accept statements and entry calls.

3.2.6 Delay Statements

Within an untimed partial correctness framework the meaning of the delay statement is simply the side-effect of the evaluation of the delay expressions.

3.2.7 Select Statements

Select statements can simply be modeled using the choice operator (+) of process logic. Select alternatives either begin with an accept statement, a when clause, or a terminate statement. In the case of an alternative beginning with an accept statement, the pr-process of the alternative is of the form (see 3.2.5):

$$self.e_\alpha?x \cdot \ldots$$

For guarded alternatives the semantics is given by

$$Wc[\![\texttt{when C =>}]\!]p = \mathsf{if}\ C\ \mathsf{then}\ p\ \mathsf{else}\ \delta$$

The pre-process of a terminate alternative is of the form $\gamma\sharp \cdot \epsilon$ (see 3.2.3).

The semantics of a select statement simply becomes a choice of the different select alternatives:

$$\begin{array}{ll} & self.e1_\alpha?x \cdot \ldots \\ + & \ldots \\ + & \mathsf{if}\ C\ \mathsf{then}\ self.en_\alpha?x \cdot \ldots\ \mathsf{else}\ \delta \\ + & \gamma\sharp \cdot \epsilon \end{array}$$

3.2.8 Conditional Entry Calls

Without modeling timing or entry queues the semantics of conditional entry calls is a non-deterministic choice between the entry call and the else part. For instance, the pre-process of a conditional entry

```
select t.e (x, y) else S
```

is

$$t.e_\alpha!x \cdot t.e_\omega?y \cdot p + \tau \cdot Wc[\![\mathtt{S}]\!]p$$

Thus, the process may choose the entry call or choose to proceed with the else part. The latter is a possible choice even if a rendezvous with $t.e$ is possible. This is sound because it is not observable in our framework whether or not a rendezvous is possible.

3.2.9 Abort Statements

As it stands, the definition does not handle abort statements. Describing general abort semantics in a process logic framework is rather difficult. The problem is that abortion is an action by one process that affect a second process without its cooperation.

A possible definition might involve the addition of abort alternatives to all communications that task can perform. But such an approach would not faithfully model the fact that an infinite loop (without communication) can be terminated with an abort statement.

While this is not a problem in a partial correctness framework, it makes it much more difficult to add termination proofs. In the present semantics, termination can be shown using standard well-founded ordering techniques on all loops and recursive subprograms. This will be no longer the case if abort statements are defined in the manner described above because a loop may terminate due to an abort statement in a different process.

A better solution may be to extend process logic with a disabling operator akin to that of LOTOS ([22]).

3.3 Annotations and Proofs

Given the kind of predicate transformers described above, the question arises how the programmer should construct annotations for a program. This section discusses some of these issues and compares the situation with the sequential case.

3.3.1 Invariants in Concurrent Programs

Consider the predicate transformers for a sequential while loop of the form:

`while` b `loop` S

with postcondition q. In some suitably rich language one could define the weakest pre-process of the loop as

$$Wp^*[\![\text{while } b \text{ loop } S]\!]q = \mu r.(r \rightarrow \text{if } b \text{ then } Wp[\![S]\!]r \text{ else } q)$$

Since ordinary first-order logic does not contain a fixed point operator μ this formulation is not very practical[5]. Instead, the typical solution is to use user-provided invariants to define an approximation to Wp^*. The idea is that if for some arbitrary formula I one can prove that

$$I \rightarrow \text{if } b \text{ then } Wp[\![S]\!]I \text{ else } q$$

then monotonicity of predicate transformers with respect to $\rightarrow$ guarantees that I is larger (stronger) than that the least (weakest) fixed point, i.e.

$$I \rightarrow Wp^*[\![\text{while } b \text{ do } S]\!]q$$

Obviously, the use of loop invariants is convenient for sequential programs. It allows proofs to use first-order logic and it associates induction hypotheses with the loop they relate to[6].

The situation is slightly more complex in the concurrent case. Consider the producer and a consumer tasks given by

$$\begin{aligned} p(n) &::= \alpha!n \cdot \text{if } n > 0 \text{ then } p(n-1) \text{ else } \epsilon \\ q(m) &::= \alpha?x \cdot \text{if } m > 0 \text{ then } q(m-1) \text{ else } \epsilon \end{aligned}$$

Both of these processes (actually their respective tasks) contain a loop that may have an invariant. Suppose one wants to prove that the process

$$\partial\{\alpha\}.(p(k) \mid q(k))$$

will not deadlock. This illustrates the following problem: The absence of deadlock is a global property. It depends on the global invariant that m and n are decremented in lock step and that there are no other processes that send or receive α. A local invariant cannot help to establish this property.

Thus, in the concurrent case loops that involve communication cannot be adequately described by local invariants. In effect, suitable invariants need to be provided as induction hypotheses at proof time once all of the global context is known (see section 4.2 for an example). This poses no conceptual problem since appropriate pre-processes (fixed points) can be expressed in process logic. The difficulty is one of human engineering: it is much simpler and more intuitive for the user to provide induction hypotheses locally with the loop rather than at proof time when the connection to the source has been lost. It may be appropriate to introduce new kinds of annotations that are associated with collections of tasks (e.g. all tasks with a common master). Such annotations could be used to express invariants that cross multiple task boundaries.

In process logic, loop invariants are optional. For the loop

[5] Dijkstra's initial formulation of weakest preconditions assumes infinite disjunctions.

[6] This seems to be a special case of reusable proof directives embedded in the program text.

```
while E loop
    S
end loop;
```

with post-process p the semantics is given by

$$\mu f.\lambda\vec{x}.\,\mathsf{if}\,E\,\mathsf{then}\,Wc[\![\mathtt{S}]\!]f(\vec{x})\,\mathsf{else}\,p$$

where $\vec{x}$ is the set of all variables changed in S. If an invariant $I(\vec{x})$ is given the pre-process is $I(\vec{x})$ with the verification condition

$$I(\vec{x}) \Rightarrow \mathsf{if}\,E\,\mathsf{then}\,Wc[\![\mathtt{S}]\!]I(\vec{x})\,\mathsf{else}\,p$$

3.3.2 Procedure Annotations

The execution of a procedure is a process that is executed sequentially with the calling process. A procedure may perform arbitrary events and terminate in a particular state.

Consequently, the postcondition of a procedure is a first-order formula that characterizes the possible final state. The pre-process of a procedure is a process term that describes the process performed by the procedure depending on the initial state.

Given a procedure P with the pre-process p and the postcondition A, the pre-process of a call to P is

$$Wc[\![\mathtt{P}]\!]q = p; \Sigma_{\vec{x}}\,\mathsf{if}\,A\,\mathsf{then}\,q\,\mathsf{else}\,\delta$$

where $\vec{x}$ is the set of variables modified in the call.

3.3.3 Abstraction and Action Refinement

To deal with complex tasking programs one may wish to introduce abstraction for communication events. The classical example is the implementation of networking protocol. Typically, protocols are specified in layers. Each layer may be viewed abstractly as a defined by a process whose abstract actions (e.g. the sending of a message) are implemented through several actions in the lower layer.

Unfortunately, the abstraction of several actions into new, more abstract ones and the refinement of actions are not sound operations in general. The problem is that there is no guarantee that lower level actions are performed in proper order and that there is no interference from other processes. It is worthwhile to study conditions under which abstraction and refinement will be legal.

One important special case arises in Ada. If an entry call does not raise exceptions and the associated accept does not contain nested accepts or entry calls the complete entry call can

be represented as a single action. The following shorthand notation will be used in this case:

$$t.e?x!e \cdot p \quad \text{means} \quad t.e_\alpha?x \cdot t.e_\omega!e \cdot p$$
$$t.e!e?x \cdot p \quad \text{means} \quad t.e_\alpha!e \cdot t.e_\omega?x \cdot p$$

The former is the action of an accept, the latter is the action of a matching call.

Section 4 shows some realistic examples that require more complex abstract actions to be manageable.

4 Examples

4.1 A Buffer Task

4.1.1 The Program

The following example is based on the buffer task given in the Ada LRM [6, (9.12)].

```
task BUFFER is
   --| behavior P1
   entry READ (C : out CHARACTER);
   entry WRITE(C : in  CHARACTER);
end;

task body BUFFER is
   POOL_SIZE : constant INTEGER := 1
   POOL      : array(1 .. POOL_SIZE) of CHARACTER;
   COUNT     : INTEGER range 0 .. POOL_SIZE := 0;
   IN_INDEX, OUT_INDEX : INTEGER range 1 .. POOL_SIZE := 1;
begin
   --| P2
   loop
      --| invariant P3
      select
         --| P4.1
         when COUNT < POOL_SIZE =>
            accept WRITE(C : in CHARACTER) do
               POOL(IN_INDEX) := C;
            end;
            IN_INDEX := IN_INDEX mod POOL_SIZE + 1;
            COUNT    := COUNT + 1;
      or
         --| P4.2
```

```
            when COUNT > 0 =>
               accept READ(C : out CHARACTER) do
                  C := POOL(OUT_INDEX);
               end;
               OUT_INDEX := OUT_INDEX mod POOL_SIZE + 1;
               COUNT     := COUNT - 1;
         or
            --| P4.3
            terminate;
         end select;
      end loop;
      --| P5
   end BUFFER;
```

Annotations `Pi` are defined below. Annotations `P4.j` and `P2` are optional. They are included here for illustration purposes.

4.1.2 Annotations

Even for this seemingly simple program it is indispensable to introduce appropriate abstractions. The semantics of the buffer can best be expressed in terms of the sequence of elements buffered. We define the abstraction function σ

$$\sigma : ArrayIntChar, Int, Int, Int \rightarrow seq(Char)$$

that maps the current value of `POOL`, `COUNT`, `IN_INDEX` and `OUT_INDEX` into the sequence of characters currently in the buffer. Note that the signature of σ is written using the Larch sorts corresponding to the Ada types of the program according to the rules of Penelope.

Function σ can be defined as follows:

$$\begin{aligned} \sigma(p, 0, i, o) &= \langle\rangle \\ \sigma(p, c+1, i, o) &= \langle p[o]\rangle \& \sigma(p, c, i, o \bmod p_s + 1) \end{aligned}$$

where $\langle\rangle$ denotes sequence constructor, "&" means concatenation, and p_s is the value of the constant `POOL_SIZE`.

Next, we define a suitable abstract process that captures the behavior of the buffer task.

$$\begin{aligned} B(s) \;::=\; & \text{if } length(s) < p_s \text{ then write?}c! \cdot B(s\&\langle c\rangle) \text{ else } \delta \\ & + \text{ if } s \neq \langle\rangle \text{ then read?}!first(s) \cdot B(rest(s)) \text{ else } \delta \\ & + \gamma\sharp \cdot \epsilon \end{aligned}$$

where $B(s)$ is a buffer process storing the data in sequence s and where γ is the synchronization event associated with the scope of declaration of the task. Note the use of the shorthand notation write?$c! \cdot \ldots$.

P5 is defined to be ϵ. This is reasonable since nothing about the final state is observable. Alternatively, one might wish to state that

$$\sigma(\texttt{POOL}, \texttt{COUNT}, \texttt{IN\_INDEX}, \texttt{OUT\_INDEX}) = \langle\rangle$$

requiring that the task can only terminate if the buffer is empty. But this is not true for the given task.

P3 is the loop invariant. It is defined as

$$B(\sigma(\texttt{POOL}, \texttt{COUNT}, \texttt{IN\_INDEX}, \texttt{OUT\_INDEX}))$$

This invariant states that the task behaves like an abstract buffer that contains the sequence of characters as defined by σ.

P4.1 is the pre-process generated for the write alternative with the invariant as post-process. Applying the predicate transformers yields:

$$\begin{array}{l}\textsf{if } \texttt{COUNT} < p_s \textsf{ then} \\ \quad \text{write?C!}\cdot \\ \quad (\sigma(\texttt{POOL}[\texttt{IN\_INDEX} \Rightarrow \texttt{C}]\texttt{COUNT} + 1, \texttt{IN\_INDEX } \mathbf{mod}\ p_s + 1), \texttt{OUT\_INDEX}) \\ \textsf{else } \delta\end{array}$$

P4.2 is the pre-process generated for the read alternative which is similarly determined to be

$$\begin{array}{l}\textsf{if } \texttt{COUNT} > 0 \textsf{ then} \\ \quad \text{read?!POOL[OUT\_INDEX]}\cdot \\ \quad\quad B(\sigma(\texttt{POOL}, \texttt{COUNT} - 1, \texttt{IN\_INDEX}, \texttt{OUT\_INDEX } \mathbf{mod}\ p_s + 1)) \\ \textsf{else } \delta\end{array}$$

P4.3 is the pre-process of the terminate alternative. This is simply $\gamma\sharp \cdot \epsilon$.

P2 is the pre-process of the loop and is identical to P3, the loop invariant.

P1 describes the behavior of the task given by $B(\langle\rangle)$. It is constructed from the invariant P12 by accounting for the initializations in the declarative part of the task.

4.1.3 Verification Conditions

The pre-process of the select statement is just the alternative selection of the pre-processes of the three select alternatives, i.e.

$$\begin{array}{l}\textsf{if } \texttt{COUNT} < p_s \textsf{ then} \\ \quad \text{write?C!}\cdot \\ \quad\quad B(\sigma(\texttt{POOL}[\texttt{IN\_INDEX} \Rightarrow \texttt{C}], \\ \quad\quad\quad \texttt{COUNT} + 1, \\ \quad\quad\quad \texttt{IN\_INDEX } \mathbf{mod}\ p_s + 1, \\ \quad\quad\quad \texttt{OUT\_INDEX})) \\ \textsf{else } \delta\end{array}$$

$$
\begin{array}{l}
+ \quad \textsf{if}\ \texttt{COUNT} > 0\ \textsf{then} \\
\qquad \texttt{read?!POOL[OUT\_INDEX]} \cdot \\
\qquad\quad B(\sigma(\texttt{POOL}, \texttt{COUNT} - 1, \texttt{IN\_INDEX}, \texttt{OUT\_INDEX}\ \mathbf{mod}\ p_s + 1)) \\
\qquad \textsf{else}\ \delta \\
+ \quad \gamma\sharp \cdot \epsilon
\end{array}
$$

which leads to the following verification condition for the loop

$$
\begin{array}{l}
B(\sigma(\texttt{POOL}, \texttt{COUNT}, \texttt{IN\_INDEX}, \texttt{OUT\_INDEX})) \\
\Rightarrow \\
\quad \textsf{if}\ \texttt{COUNT} < p_s\ \textsf{then} \\
\qquad \texttt{write?C!} \cdot \\
\qquad\quad B(\sigma(\texttt{POOL[IN\_INDEX} \Rightarrow \texttt{C}], \\
\qquad\qquad \texttt{COUNT} + 1, \\
\qquad\qquad \texttt{IN\_INDEX}\ \mathbf{mod}\ p_s + 1, \\
\qquad\qquad \texttt{OUT\_INDEX})) \\
\quad \textsf{else}\ \delta \\
\quad + \ \textsf{if}\ \texttt{COUNT} > 0\ \textsf{then} \\
\qquad\quad \texttt{read?!POOL[OUT\_INDEX]} \cdot \\
\qquad\qquad B(\sigma(\texttt{POOL}, \texttt{COUNT} - 1, \texttt{IN\_INDEX}, \texttt{OUT\_INDEX}\ \mathbf{mod}\ p_s + 1)) \\
\qquad\quad \textsf{else}\ \delta \\
\quad + \gamma\sharp \cdot \epsilon
\end{array}
$$

To prove this approximation observe the following equalities that follow directly from the abstraction function (σ):

$$
\begin{array}{l}
\mathit{length}(\sigma(\texttt{POOL}, \texttt{COUNT}, \texttt{IN\_INDEX}, \texttt{OUT\_INDEX})) = \texttt{COUNT} \\
\\
\texttt{C}\&\sigma(\texttt{POOL}, \texttt{COUNT}, \texttt{IN\_INDEX}, \texttt{OUT\_INDEX}) \\
\quad = \sigma(\texttt{POOL[IN\_INDEX} \Rightarrow \texttt{C}], \texttt{COUNT} + 1, \texttt{IN\_INDEX}\ \mathbf{mod}\ p_s + 1, \texttt{OUT\_INDEX}) \\
\\
(\sigma(\texttt{POOL}, \texttt{COUNT}, \texttt{IN\_INDEX}, \texttt{OUT\_INDEX}) \neq \langle\rangle) = (\texttt{COUNT} > 0) \\
\\
\mathit{first}(\sigma(\texttt{POOL}, \texttt{COUNT}, \texttt{IN\_INDEX}, \texttt{OUT\_INDEX})) = \texttt{POOL[OUT\_INDEX]} \\
\\
\mathit{rest}(\sigma(\texttt{POOL}, \texttt{COUNT}, \texttt{IN\_INDEX}, \texttt{OUT\_INDEX})) \\
\quad = \sigma(\texttt{POOL}, \texttt{COUNT} - 1, \texttt{IN\_INDEX}, \texttt{OUT\_INDEX}\ \mathbf{mod}\ p_s + 1)
\end{array}
$$

Next, write s for $\sigma(\texttt{POOL}, \texttt{COUNT}, \texttt{IN\_INDEX}, \texttt{OUT\_INDEX})$ and substitute in the verification condition:

$$
\begin{array}{l}
B(s) \\
\quad \Rightarrow \\
\qquad \textsf{if}\ \mathit{length}(s) < p_s\ \textsf{then}\ \texttt{write?C!} \cdot B(\texttt{C}\&s)\ \textsf{else}\ \delta \\
\qquad + \ \textsf{if}\ \mathit{length}(s) > 0\ \textsf{then} \\
\qquad\quad \texttt{read?!}\mathit{first}(s) \cdot B(\sigma(\mathit{rest}(s)) \\
\qquad\quad \textsf{else}\ \delta \\
\qquad + \gamma\sharp \cdot \epsilon
\end{array}
$$

Which follows directly from the definition of B. More precisely, since B is defined by $B(\ldots) ::= p$ it follows that $B(\ldots) = p$ and thus $B(\ldots) \Rightarrow p$.

A second verification condition is generated to prove that P1 implies the pre-process of the task body, i.e.

$$B(\langle\rangle) \Rightarrow B(\sigma(\texttt{POOL}, 0, 1, 1))$$

which follows trivially since $\sigma(\texttt{POOL}, 0, 1, 1) = \langle\rangle$.

4.2 Multi-Set Partitioning

In the following example a process specification is developed from an informal problem statement. It is shown how an Ada tasking program together with its annotations can be derived from this formal specification. The example demonstrates the systematic development of a program from specifications and the use of the formalism as a guide to the implementation in the style advocated by Gries ([50]). The importance is that process logic enables us to perform such derivations for concurrent programs in a natural way.

The problem to be solved is a variant of one proposed by Dijkstra in [40, pages 260, 261]. Given are two tasks that each locally store a non-empty multi-sets of numbers. The problem is for the two tasks to exchange numbers until all numbers in one of the multi-sets are greater or equal to those in the other.

4.2.1 Design

The problem can be formalized in process logic as follows. Let $high(m_1)$ and $low(m_2)$ be two processes whose local variables are m_1 and m_2 respectively. A first approach at defining the behavior of *high* might be to define

$$high(m) ::= \alpha!min(m) \cdot \beta?x \cdot (\textsf{if}\ x \leq min(m)\ \textsf{then}\ \epsilon\ \textsf{else}\ high(m + x - min(m)))$$

where $min(m)$ is the minimum element of m and $+$ and $-$ denote the addition and removal of an element from a multi-set. Since it is assumed that m is non-empty, the *min* function is well defined. By symmetry, *low* can be defined as

$$low(m) ::= \alpha?y \cdot \beta!max(m) \cdot (\textsf{if}\ y \geq max(m)\ \textsf{then}\ \epsilon\ \textsf{else}\ low(m + y - max(m)))$$

Assuming that it is possible to implement two tasks with the given behavior, we can attempt to prove on the specification level that these two tasks will interact properly and solve the problem. We are interested in the parallel execution of $high(m_1)$ and $low(m_2)$. But as has been shown in earlier examples, the term $high(m_1) \mid low(m_2)$ will lead to a combinatorial explosion of terms, most of which are uninteresting. So, what we are really interested in is $\partial\{\alpha, \beta\}.(high(m_1) \mid low(m_2))$. If we evaluate this latter term we get:

$$\partial\{\alpha, \beta\}.(high(m_1) \mid low(m_2))$$

$$
\begin{aligned}
&= \partial\{\alpha,\beta\}.\ \beta?x \cdot (\textsf{if } x \le min(m_1) \textsf{ then } \epsilon \textsf{ else } high(m_1 + x - min(m_1))) \\
&\qquad\qquad \mid \beta!max(m_2) \cdot \ \textsf{if } min(m_1) \ge max(m_2) \textsf{ then } \epsilon \\
&\qquad\qquad\qquad\qquad \textsf{else } low(m_2 + min(m_1) - max(m_2)) \\
&= \partial\{\alpha,\beta\}.\ (\textsf{if } max(m_2) \le min(m_1) \textsf{ then } \epsilon \textsf{ else} \\
&\qquad\qquad high(m_1 + max(m_2) - min(m_1))) \\
&\qquad\qquad \mid (\textsf{if } min(m_1) \ge max(m_2) \textsf{ then } \epsilon \textsf{ else} \\
&\qquad\qquad low(m_2 + min(m_1) - max(m_2))) \\
&= \textsf{if } max(m_2) \le min(m_1) \textsf{ then } \epsilon \textsf{ else} \\
&\quad \partial\{\alpha,\beta\}.\ (high(m_1 + max(m_2) - min(m_1))) \\
&\qquad\qquad \mid low(m_2 + min(m_1) - max(m_2))
\end{aligned}
$$

This term says that the parallel execution of $high(m_1)$ and $low(m_2)$ will terminate when $max(m_2) \le min(m_1)$ and will otherwise continue as the process $high(m_1 + max(m_2) - min(m_1)) \mid low(m_2 + min(m_1) - max(m_2))$, i.e., is a state where both tasks have made progress towards the solution. The process solves the problem since it makes no assumptions about the initial state (to be correct one would need to add the explicit conditions that neither bag is empty). This follows since one can show

$$\epsilon \Rightarrow \partial\{\alpha,\beta\}.(high(m_1) \mid low(m_2))$$

Note that ϵ is the strongest observable behavior since the processes do not communicate the results of their computation.

Next the specifications are altered slightly to fit Ada tasks. First, the initial and final values need to be communicated through entry calls. This can be done by specifying

$$
\begin{aligned}
proc_h &= init?m \cdot high(m) \\
proc_l &= init?m \cdot low(m) \\
high(m) &::= \alpha!min(m) \cdot \beta?x \cdot \ \textsf{if } x \le min(m) \textsf{ then} \\
&\qquad\qquad\qquad\qquad result!m \cdot \epsilon \\
&\qquad\qquad\qquad\qquad \textsf{else} \\
&\qquad\qquad\qquad\qquad high(m + x - min(m)) \\
low(m) &::= \alpha?y \cdot \beta!max(m) \cdot \ \textsf{if } y \ge max(m) \textsf{ then} \\
&\qquad\qquad\qquad\qquad result!m \cdot \epsilon \\
&\qquad\qquad\qquad\qquad \textsf{else} \\
&\qquad\qquad\qquad\qquad low(m + y - max(m))
\end{aligned}
$$

To develop the Ada code, it must be decided how entry calls are used to perform the actual communication. Obviously, *init* should become an entry to the tasks; *result* could be a call to some third tasks or it could be another entry that returns the actual result through an out parameter. The interesting cases are the events α and β. These could be implemented as an entry each or, alternatively, α and β could be made the start and end of a single rendezvous. Choosing the latter alternative the final specification for the two tasks is

$$proc_h = init?m! \cdot high(m)$$

$$
\begin{aligned}
proc_l &= init?m! \cdot low(m) \\
high(m) &::= t_l.swap!min(m)?x \cdot \\
&\qquad \textsf{if } x \leq min(m) \textsf{ then } result?x!m \cdot \epsilon \textsf{ else} \\
&\qquad\quad high(m + x - min(m)) \\
low(m) &::= swap?y!max(m) \cdot \\
&\qquad \textsf{if } y \geq max(m) \textsf{ then } result?y!m \cdot \epsilon \textsf{ else} \\
&\qquad\quad low(m + y - max(m))
\end{aligned}
$$

4.2.2 The Program

The translation of this specification into Ada tasks is now straightforward. First, the two task specifications are:

```
task TH is
    entry init (m : multi_set);
    entry result (m : out multi_set);
end TH;

task TL is
    entry init (m : multi_set);
    entry swap (y : integer; x : out integer);
    entry result (m : out multi_set);
end TL;
```

The initialization part of the tasks follows directly from the specification. The processes *high* and *low*, however, need to be translated into two loops. This is a trivial transformation, since the specification is in some sense "tail recursive". The two tasks receive their initial values, communicate through the *swap* entry and return the final result.

```
task body TH is
    bh : multi_set;
    mi, mx : integer;
begin
    accept init (m : multi_set) do
        bh := m;
    end accept;

    loop
        --| invariant high (bh);
        mi := min (bh);
        TL.swap (mi, mx);
```

```
        exit when mx <= mi;
        bh := bh + mx - mi;
    end loop;

    accept result (m : out multi_set) do
        m := bh;
    end accept;
end TH;


task body TL is
    bl : multi_set;
    mx, mi : integer;
begin
    accept init (m : multi_set) do
        bl := m;
    end accept;

    loop
        --| invariant low (bl)
        mx := max (bl);
        accept swap (y : integer; x : out integer) do
            mi := y;
            x := mx;
        end accept;
        exit when mx <= mi;
        bl := bl + mi - mx;
    end loop;

    accept result (m : out multi_set) do
        m := bl;
    end accept;
end TL;
```

4.2.3 Verification Conditions

Using the rules in [111] and the loop rule above, the pre-process of task TH for post-process ϵ becomes

$$init?b_h! \cdot high(b_h)$$

where $high(b_h)$ is the loop invariant given in TH as defined above. The verification condition for the loop is computed as follows. With post-process ϵ the pre-process of the final accept statement is

$$result?!b_h \cdot \epsilon$$

which becomes the post-process of the loop. The loop verification condition shows that the given invariant is preserved by the body of the loop, i.e.

$$
\begin{array}{l}
high(b_h) \Rightarrow \\
\quad t_l.swap!min(b_h)?mx\cdot \\
\quad\quad \textsf{if } mx \leq min(b_h) \textsf{ then } result?!b_h \cdot \epsilon \textsf{ else } high(b_h + mx - min(b_h))
\end{array}
$$

To simplify matters slightly, the synchronization events used to describe task termination have been omitted here.

Similarly, for task TL we get the pre-process

$$init?b_l! \cdot low(b_l)$$

with the verification condition

$$
\begin{array}{l}
low(b_l) \Rightarrow \\
\quad swap?x!max(b_l)\cdot \\
\quad\quad \textsf{if } max(b_l) \leq x \textsf{ then } result?!b_l \cdot \epsilon \textsf{ else } low(b_l + x - max(b_l))
\end{array}
$$

Both of these verification conditions follow immediately from the definitions of *low* and *high* above.

4.2.4 A Calling Environment

The following procedure is an application of the partitioning solution: the two task instances are locally declared in procedure *partition*. The objective is to prove that *partition* satisfies a conventional sequential specification.

```
procedure partition (b1, b2 : multi_set;
                     h, l : out multi_set) is
--| in not empty (b1) and not empty (b2)
--| out l = fl (b1, b2) and h = fh (b1, b2);
    TaskL : tl;
    TaskH : th;
begin
    TaskH.init (b1);
    TaskL.init (b2);
    TaskH.result (h);
    TaskL.result (l);
end
```

We write $P_{in}(b_1, b_2)$ and $P_{out}(h, l, b_1, b_2)$ for the specifications of the procedure. Functions f_l and f_h are defined as

$$
\begin{array}{l}
f_l(b_1, b_2) ::= \\
\quad \textsf{if } max(b_2) \leq min(b_1) \textsf{ then } b_2 \\
\quad \textsf{else } f_l(b_1 + max(b_2) - min(b_1), b_2 + min(b_1) - max(b_2))
\end{array}
$$

$$
\begin{array}{l}
f_h(b_1, b_2) ::= \\
\quad \textsf{if } max(b_2) \leq min(b_1) \textsf{ then } b_1 \\
\quad \textsf{else } f_l(b_1 + max(b_2) - min(b_1), b_2 + min(b_1) - max(b_2))
\end{array}
$$

The verification condition for procedure `Partition` says that the entry condition $P_{in}(b_1, b_2)$ "implies" the pre-process of the body, more precisely:

$$
\begin{array}{l}
\textsf{if } P_{in}(b_1, b_2) \textsf{ then } \epsilon \textsf{ else } \top \Rightarrow \\
\partial\{t_h, t_l\}.t_l.init?b_l! \cdot L(b_l) \\
\quad \mid t_h.init?b_h! \cdot H(b_h) \\
\quad \mid t_h.init!b_1? \cdot t_l.init!b_2? \cdot t_h.result!?h \cdot t_l.result!?l \cdot \\
\quad\quad \textsf{if } P_{out}(h, l, b_1, b_2) \textsf{ then } \epsilon \textsf{ else } \top
\end{array}
$$

Where $L = low[t_l/self]$ and $H = high[t_h/self]$ are the processes defining the two instances of `TL` and `TH` and t_h and t_l are new unique names for these instances introduced by the VC generator. The notation $\partial\{t\}.p$ is a shorthand for $\partial\{\beta_1, \ldots, \beta_n\}.p$ for all events β_i associated with task t.

There are some obvious simplifications that can be applied to the right-hand side of the approximation relation. The only possible events correspond to the rendezvous of the init entry of t_h. This leads to

$$
\begin{array}{l}
\textsf{if } P_{in}(b_1, b_2) \textsf{ then } \epsilon \textsf{ else } \top \Rightarrow \\
\partial\{t_h, t_l\}.\tau \cdot t_l.init?b_l! \cdot L(b_l) \\
\quad \mid H(b_1) \\
\quad \mid t_l.init!b_2? \cdot t_h.result!?h \cdot t_l.result!?l \cdot \textsf{if } P_{out}(h, l, b_1, b_2) \textsf{ then } \epsilon \textsf{ else } \top
\end{array}
$$

Similarly, only the rendezvous with $[t_l.init]$ is possible. This simplification is somewhat less obvious since it requires the unfolding of the definition of H to determine that process H cannot engage in a rendezvous with the *init* entry ot L.

$$
\begin{array}{l}
\textsf{if } P_{in}(b_1, b_2) \textsf{ then } \epsilon \textsf{ else } \top \Rightarrow \\
\quad \partial\{t_h, t_l\}.\tau \cdot (L(b_2) \mid H(b_1) \\
\quad\quad \mid t_h.result!?h \cdot t_l.result!?l \cdot \textsf{if } P_{out}(h, l, b_1, b_2) \textsf{ then } \epsilon \textsf{ else } \top
\end{array}
$$

Using the definition of the hiding operator, the right hand side is equal to

$$
\begin{array}{l}
\partial\{[t_l.result], [t_h.result]\}.\tau \cdot \\
\quad \partial\{[t_l.swap]\}.\,(L(b_2) \mid H(b_1)) \\
\quad \mid t_h.result!?h \cdot t_l.result!?l \cdot \textsf{if } P_{out}(h, l, b_1, b_2) \textsf{ then } \epsilon \textsf{ else } \top
\end{array}
$$

The process

$$\partial\{[t_l.swap]\}.(L(b_2) \mid H(b_1))$$

represents a global loop that spans two tasks. In essence, TL and TH execute in lock step. We use a global invariant to reason about this term. It will be shown that

$$\begin{aligned} & t_l.result?!f_l(b_1,b_2)\cdot\epsilon \mid t_h.result?!f_h(b_1,b_2)\cdot\epsilon \\ & \quad \Rightarrow \partial\{[t_l.swap]\}.(L(b_2) \mid H(b_1)) \qquad (*) \end{aligned}$$

Once (*) has been shown, the correctness of the procedure follows immediately: because of monotonicity the verification condition reduces to

$$\begin{aligned} & \mathsf{if}\ P_{in}(b_1,b_2)\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top \Rightarrow \\ & \ \partial\{[t_l.result],[t_h.result]\}.\tau\cdot \\ & \quad t_l.result?!f_l(b_1,b_2)\cdot\epsilon \mid t_h.result?!f_h(b_1,b_2)\cdot\epsilon \\ & \quad \mid t_h.result!?h\cdot t_l.result!?l\cdot \mathsf{if}\ P_{out}(h,l,b_1,b_2)\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top \end{aligned}$$

which simplifies to

$$\begin{aligned} & \mathsf{if}\ P_{in}(b_1,b_2)\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top \\ & \quad \Rightarrow \epsilon \mid \epsilon \mid \mathsf{if}\ P_{out}(f_h(b_1,b_2),f_l(b_1,b_2),b_1,b_2)\ \mathsf{then}\ \epsilon\ \mathsf{else}\ \top \end{aligned}$$

and follows from the definition of P_{out}.

It remains to be shown that (*) holds. Expanding the definitions for L and H we get

$$\begin{aligned} & \partial\{[t_l.swap]\}.(H(b_1) \mid L(b_2)) = \\ & \partial\{[t_l.swap]\}. \\ & \ t_l.swap!min(b_1)?mx\cdot \\ & \quad \mathsf{if}\ mx \le min(b_1)\ \mathsf{then}\ t_h.result?!b_1\cdot\epsilon\ \mathsf{else}\ H(b_1+mx-min(b_1)) \\ & \ \mid t_l.swap?x!max(b_2)\cdot \\ & \quad \mathsf{if}\ max(b_2)\le x\ \mathsf{then}\ t_l.result?!b_2\cdot\epsilon\ \mathsf{else}\ L(b_2+x-max(b_2)) \end{aligned}$$

Because of the hiding of entry *swap* no outside process can interfere and the concurrent composition simplifies to

$$\begin{aligned} & \ldots = \\ & \partial\{[t_l.swap]\}.\tau\cdot \\ & \ \mathsf{if}\ max(b_2)\le min(b_1)\ \mathsf{then}\ t_h.result?!b_1\cdot\epsilon\ \mathsf{else}\ H(b_1+max(b_2)-min(b_1)) \\ & \mid \mathsf{if}\ max(b_2)\le min(b_1)\ \mathsf{then}\ t_l.result?!b_2\cdot\epsilon\ \mathsf{else}\ L(b_2+min(b_1)-max(b_2)) \end{aligned}$$

Finally, this process can be transformed by using case analysis. In effect, the property $p = \mathsf{if}\ A\ \mathsf{then}\ p\ \mathsf{else}\ p$ is used here. The two instances of p can be simplified based on the guard predicate. In this case $A = max(b_2) \le min(b_1)$ gives

$$\begin{aligned} & \partial\{[t_l.swap]\}.(H(b_1) \mid L(b_2)) = \\ & \ \mathsf{if}(max(b_2)\le min(b_1))\ \mathsf{then} \\ & \quad \tau\cdot(t_h.result?!b_1\cdot\epsilon \mid t_l.result?!b_2\cdot\epsilon) \\ & \ \mathsf{else} \\ & \quad \tau\cdot\partial\{[t_l.swap]\}. \\ & \quad (H(b_1+max(b_2)-min(b_1)) \mid L(b_2+min(b_1)-max(b_2))) \end{aligned}$$

Thus, if it can be shown that

$$
\begin{array}{l}
t_l.result?!f_l(b_1,b_2)\cdot\epsilon \mid t_l.result?!f_h(b_1,b_2)\cdot\epsilon \Rightarrow \\
\quad \mathsf{if}(max(b_2) \leq min(b_1))\,\mathsf{then} \\
\quad\quad \tau\cdot(t_h.result?!b_1\cdot\epsilon \mid t_l.result?!b_2\cdot\epsilon) \\
\quad \mathsf{else} \\
\quad\quad \tau\cdot t_l.result?!f_l(b_1+max(b_2)-min(b_1),b_2+min(b_1)-max(b_2))\cdot\epsilon \\
\quad\quad\quad \mid t_h.result?!f_h(b_1+max(b_2)-min(b_1),b_2+min(b_1)-max(b_2))\cdot\epsilon
\end{array}
$$

then (*) follows by fixed point induction. Now, by the definition of f_l and f_h this is

$$
\begin{array}{l}
t_l.result?!f_l(b_1,b_2)\cdot\epsilon \mid t_h.result?!f_h(b_1,b_2)\cdot\epsilon \Rightarrow \\
\quad \mathsf{if}(max(b_2) \leq min(b_1))\,\mathsf{then} \\
\quad\quad \tau\cdot(t_h.result?!b_1\cdot\epsilon \mid t_l.result?!b_2\cdot\epsilon) \\
\quad \mathsf{else} \\
\quad\quad \tau\cdot t_l.result?!f_l(b_1,b_2)\cdot\epsilon \mid t_h.result?!f_h(b_1,b_2)\cdot\epsilon
\end{array}
$$

or

$$
\begin{array}{l}
t_l.result?!f_l(b_1,b_2)\cdot\epsilon \mid t_h.result?!f_h(b_1,b_2)\cdot\epsilon \Rightarrow \\
\quad \tau\cdot t_l.result?!f_l(b_1,b_2)\cdot\epsilon \mid t_h.result?!f_h(b_1,b_2)\cdot\epsilon
\end{array}
$$

In this example a proof of termination can be given by using the same techniques used for sequential programs. In the example the number of iterations are bounded by the cardinality of either of the bags. The program will terminate if either of the bags is finite.

Note that the proof involved only trivial reasoning about approximation ($\Rightarrow$). The interesting parts of the proof involve equivalence preserving simplifications of process terms. This appears to be the typical approach for all programs that have sequential specifications realized by concurrent implementations.

4.3 Matrix Multiplication

The following example demonstrates that simple strategies will not in general suffice for concurrency proofs. Even more so as in the sequential case it is necessary to build appropriate abstractions for the reasoning to be tractable. Additional experience with concurrency verification is needed in order to determine what kind of extensions, abbreviations, meta-theorems and so on are appropriate for process logic.

4.3.1 The Program

The program to be verified is an Ada tasking version of a multi-processor matrix multiplication algorithm. Assume the following definition:

```
package matrix is
```

```
    k : constant integer := ...;
    m : constant integer := ...;
    n : constant integer := ...;

    type ma is array (1 .. m, 1 .. n) of integer;
    type mb is array (1 .. n, 1 .. k) of integer;
    type mc is array (1 .. m, 1 .. k) of integer;

    procedure multiply (a : ma; b : mb; c : out mc);
    --| out c = a * b;
end matrix;
```

The multiplication is to be performed by an $m \times n \times k$ cube of tasks. Values of matrix a are propagated along the x dimension, values of b are propagated along the y dimension and values of the product matrix are propagated along the z dimension. Each of the tasks performs a single multiplication and addition. The situation is illustrated in figure 4.

The multiplication package can be implemented as

```
package body matrix is
    task type cell is
        entry place (x, y, z : integer);
        entry row (r : integer);
        entry col (c : integer);
        entry sum (s : out integer);
    end cell;
    type multiplier is array (1 .. k, 1 .. m, 1 .. n) of cell;
    mtx : multiplier;
    task body cell is ...;
    procedure multiply (a : ma; b : mb; c : out mc) is ...;
begin
    for i in 1 .. k loop
        for j in 1 .. m loop
            for h in 1 .. n loop
                mtx (i, j, h).place (i, j, h);
            end loop;
        end loop;
    end loop;
--| M (mtx);
end matrix;
```

Array `mtx` is the cube of cell tasks. The initialization in the package establishes a property $M(mtx)$ of the cube which informally says that if values of matrices a and b are sent to

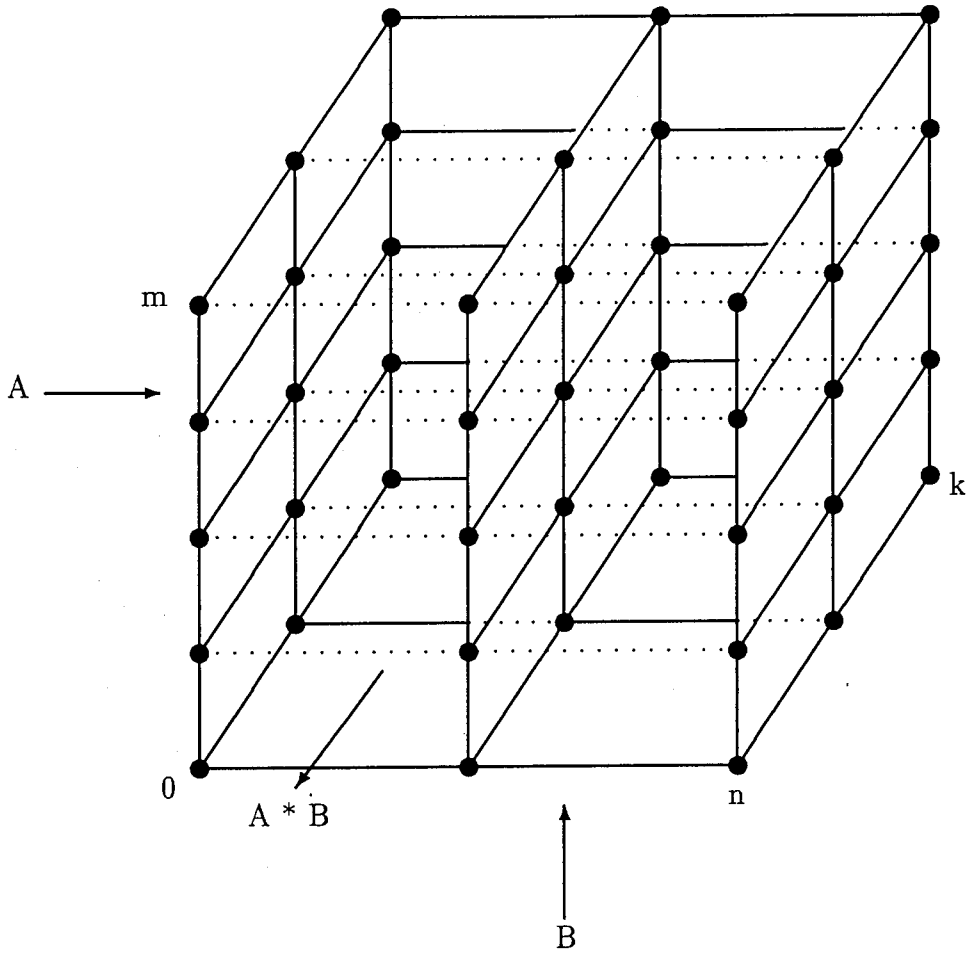

Figure 4: Task structure for matrix multiplication

the appropriate entries, the product can be read. This invariant is maintained by calls to multiply[7]. The initialization code is necessary to tell every task what its place is in the cube so it can compute its neighbors.

```
procedure multiply (a : ma; b : mb; c : out mc) is
--| global mtx;
--| in M (mtx);
--| out M (mtx) and c = a * b;
begin
    for j in 1 .. m loop for h in 1 .. n loop
        mtx (1, j, h).row(a(j,h));
    end loop; end loop;

    for i in 1 .. k loop for h in 1 .. n loop
        mtx (i, 1, h).col(b(h,i));
    end loop; end loop;

    for i in 1 .. k loop for j in 1 .. m loop
        mtx (i, j, 1).sum(c(j,i));
    end loop; end loop;
end multiply;
```

Procedure multiply initializes the $k = 1$ plane with matrix a and the $m = 1$ plane with b and reads the product from the plane $n = 1$.

The body of task cell is given as

```
task body cell is
    rr, cc, ss : integer;
    kk, mm, nn : integer;
    have_r, have_c : boolean;
begin
    accept place (x, y, z : integer) do
        kk := x; mm := y; nn := z;
    end;

    --| invariant I
    while true loop
        have_r := false; have_c := false;
        while not (have_r and have_c) loop
            select
```

[7]This invariant is actually too strong since the procedure will work properly in a concurrent environment where several tasks simultaneously call the multiplication procedure.

```
                when not have_r => accept row (r : integer) do
                    rr := r;
                end;
                if kk < k then
                    mtx(kk + 1, mm, nn).row(rr);
                end if;
                have_r := true;
            or
                when not have_c => accept col (c : integer) do
                    cc := c;
                end;
                if mm < m then
                    mtx(kk, mm + 1, nn).col(cc);
                end if;
                have_c := true;
            or
                terminate;
            end select;
        end loop;
        if nn < n then
            mtx(kk, mm, nn + 1).sum (ss);
        else
            ss := 0;
        end if;
        accept sum (s : out integer) do
            s := ss + rr * cc;
        end;
    end loop;
end cell;
```

The pre-process of the cell task can be computed as $place?\langle k, m, n\rangle \cdot I$ where process I is the invariant of the outer loop.

Because of the manner in which values are propagated the processes on the surface of the cube will behave slightly differently from those in the center. To simplify matters let us just consider the invariant for a process in the center of the cube. It will be obvious how the invariant can be modified to take care of the special boundary cases.

On close inspection is becomes clear that the inner loop will be executed exactly twice for each iteration of the outer loop. Assume that the inner loop has been unfolded - a verification condition generator can perform this trivial transformation. Thus no invariant is needed for

the inner loop and I becomes

$$\begin{aligned} I ::= & \; row?r! \cdot mtx(k+1,m,n).row!r? \cdot col?c! \cdot mtx(k,m+1,n).col!c? \cdot q \\ & + col?c! \cdot mtx(k,m+1,n).col!c? \cdot row?r! \cdot mtx(k+1,m,n).row!r? \cdot q \\ & + \gamma\sharp \cdot \epsilon \\ q ::= & \; mtx(k,m,n+1).sum!?s \cdot sum?!s + r * c \cdot I \end{aligned}$$

In the case of surface tasks, suitable conditionals need to be added to make sure that values are not passed beyond the cube, e.g. $mtx(k+1,m,n).row!r? \cdot \ldots$ only applies if $k < K$.

4.3.2 Problem Areas

This example raises a number of questions that require further study.

Task Activation An interesting problem is encountered when we consider the activation of the tasks. The present assumption is that task activation is represented in the abstract syntax. This definition is satisfactory only for the case where the number of activated tasks is statically known and unique new task values can be created by the VC generator. In the present case the array `mtx` will cause new tasks to be activated at the beginning of the block. Since the bounds of the array may vary dynamically there needs to be a representation of some form "activation loop" and a way to generate an variable number of task names and to prove that they are pair-wise distinct.

In the example one can use tuples $\langle k,m,n \rangle$ as task names such that the task at point $mtx(x,y,z)$ has the name $\langle x,y,z \rangle$. It may be possible to generalize and automate this approach. Since multiple statically declared tasks must be contained in some complex data structure, it may always be possible to use access functions (e.g. array indices or record fields) to construct suitable task names.

Another question one might ask is if process logic can express the parallel composition of a variable number of processes. The answer is yes, as the simple recursive process shows:

$$create(j) ::= \mathsf{if}\, j = 0\, \mathsf{then}\, \epsilon\, \mathsf{else}\, (p[j/self] \mid create(j-1))$$

Abstraction The matrix example makes the need for powerful *process abstractions* painfully obvious. Consider the body of procedure `multiply`. The pre-process of the loop

```
for j in 1 .. m loop for h in 1 .. n loop
    mtx (1, j, h).row(a(j,h));
end loop; end loop;
```

for post-process q is

$$e_1!a_i? \cdot \ldots e_q!a_q? \cdot q$$

for some sequence of entries e_i. It would simplify matters significantly, if it were possible to describe this whole sequence of rendezvous as a single rendezvous

$$e!a? \cdot q$$

for some abstract entry e that accepts the array a. In this particular example, informal reasoning suggests that this is indeed true. But there is no formal justification for this at present. The required reasoning appears to be much more subtle than that needed to justify collapsing the start and end events of certain special cases of entry calls.

Ultimately, it should be possible to describe the behavior of the tasks in the cube by a simple process like

$$cube = row?a \cdot col?b \cdot result!a * b \cdot cube$$

Information Hiding Related to abstraction is the issue of information hiding. In the given example one would like to understand the package `matrix` as a sequential program. The fact that there are local tasks should not be observable and should not have to be considered in writing specifications for programs using the package. The idea would be to treat every call to procedure `multiply` as a separate process that interacts with the local tasks and terminates when the call is completed. No actions performed while executing `multiply` should be observable outside the package. The formal basis for such a mechanism is unclear.

5 Conclusions

The paper has outlined a formalism for specification and verification of Ada tasking programs.

A number of areas need further exploration before proofs of concurrent programs become practical. Of particular interest are global program annotations, proof procedures, and action refinement. Progress in these areas is interdependent and requires study of large realistic examples.

The issue of global program annotations concerns ways to describe invariants that span multiple tasks in the program text. Such a mechanism will likely reduce interactions with the theorem prover and will make possible the replay of proofs after the modification of programs.

In [85] it is noted that neither weak nor strong bisimulation is decidable for CCS. Other more restrictive flavors of process algebra are decidable. For example, *Basic Process Algebra* (BPA) consists of recursively defined processes that includes sequencing and choice (but not parallel composition). Strong bisimulation is decidable for BPA ([31]).

Process logic as defined here is at least as hard as first-order logic. It needs to be investigated if and how existing process algebra provers can be adapted to deal with process logic. In

either case, one may wish to factor proofs into first-order parts and process algebra parts.

As mentioned earlier, action refinement is not generally sound. But clearly, there are special situation where action refinement is valid. The case of simple entry calls where the event starting a rendezvous is always followed by the event ending the rendezvous with the same process is one example. It needs to be investigated under which conditions action refinement is sound. Special annotation constructs need to make such refinement accessible where legal.

6 Bibliography

[1] M. Abadi, L. Lamport, Composing Specifications, TOPLAS 15(1), 1993

[2] Samson Abramsky. *A Domain Equation for Bisimulation. Information and Computing*, 92(2):161–218, June 1991.

[3] Samson Abramsky. *Observation Equivalence as a Testing Equivalence. Theoretical Computer Science*, 53:225–241, 1987.

[4] Samson Abramsky. *Tutorial on Concurrency.* 1989. Slides from an invited lecture at POPL '89.

[5] L. Aceto, M. Hennessy, Termination, Deadlock and Divergence, 5th Intl. Conf. on Mathematical Foundations of Programming Semantics, New Orleans, 1989, LNCS 442

[6] Ada Programming Language, ANSI/MIL-STD-1815A

[7] B. Alpern, F. B. Schneider, Defining liveness, Inf. Proc. Letters 21(4), 1985

[8] K. R. Apt (ed.) Logics and models of concurrent systems, Nato ASI series F, Vol 13, Springer Verlag 1985

[9] K. R. Apt, N. Francez, and W. P. de Roever. *A proof system for communicating processes. ACM TOPLAS*, 2(3):359–385, 1980.

[10] E. Astesiano, A. Giovini, and G. Reggio. *Generalized Bisimulation in Relational Specifications.* In *LNCS 294: Proceedings of STACS 88*, pages 207–226, Springer-Verlag, Berlin, 1988.

[11] E. Astesiano and E. Zucca. *Parametric Channels in CCS and Their Applications.* In *Proceedings of the 2nd Conference on Foundations of Software Technology and Theoretical Computer Science*, December 1982. Treat channels passing by encoding it into pure CCS.

[12] Egidio Astesiano and Gianna Reggio. *SMoLCS-Driven Concurrent Calculi.* In *LNCS 249: TAPSOFT '87*, pages 169–201, Springer-Verlag, Berlin, 1987.

[13] E. Astesiano, G. Reggio, Comparing direct and continuation semantics styles for concurrent languages, LNCS 247, pp. 311–322

[14] J. C. M. Baeten (ed), Applications of Process Algebra, Cambridge University Press, 1990

[15] J. C. M. Baeten, W. P. Weijland, Process Algebra, Cambridge University Press, 1990

[16] J. C. M. Baeten, J. A. Bergstra, Discrete Time Process Algebra, CONCUR 92, LNSC 630

[17] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. *Ready Trace Semantics for Concrete Process Algebra with Priority Operator.* Technical Report CWI Report CS-R8517, CWI, Amsterdam, 1985.

[18] J. A. Bergstra and J. W. Klop. *Process Algebra for Synchronous Communication. Information and Control*, 60:109–137, 1984.

[19] J. A. Bergstra and J. W. Klop, Algebra of communicating processes with abstraction, Theoretical Computer Science, 37, pp77-121, 1985

[20] David B. Benson, Jerzy Tiuryn, Fixed points in process algebras with internal actions, LNCS 239, pp53–58

[21] Bard Bloom, Sorin Istrail, and Albert Meyer. *Bisimulation Can't Be Traced.* Technical Report TR-90-1150, Cornell University, 1990. Revised version of 1988 POPL paper.

[22] T. Bolognesi and E. Brinksma. *Introduction to the ISO Specification Language LOTOS.* In *Computer Networks and ISDN Systems 14*, pages 25–59, North-Holland, Amsterdam, 1987.

[23] T. Bolognesi, M. Caneve, Squiggle - A Tool for the Analysis of LOTOS Specifications, in *Formal Description Techniques* (K. Turner, ed.) North Holland, 1989, pp201-216

[24] G. Boudol, et. al., Process Calculi, from Theory to Practice: Verification Tools, in LNCS 407, 1990

[25] Gerard Boudol and Gerard Berry. *The Chemical Abstract Machine.* In *Proceedings of the Seventeenth Annual ACM Conference on Principles of Programming Languages*, pages 81–94, ACM, January 1990.

[26] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. *A Theory of Communicating Sequential Processes. Journal of the ACM*, 31(3):560–599, July 1984.

[27] A. De Bruin, W. Böhm, The Denotational Semantics of Dynamic Networks of Processes, ACM Transactions on Programming Languages and Systems, Vol 7, No 4,, pp 656-679, October 1985

[28] S. D. Brookes, C. A. R. Hoare and A. W. Roscoe, A theory of communicating sequential processes, Journal ACM 31, pp 560-569, 1984

[29] Alan Burns, Andrew M. Lister, and Andrew J. Wellings. *A Review of Ada Tasking.* Volume 262 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1987.

[30] Ilaria Castellani and Matthew Hennessy. *Distributed Bisimulations. Journal of the ACM*, 36(4):887–911, 1989.

[31] S. Christensen, H. Hüttel, C. Stirling, Bisimulation Equivalence is Decidable for all Context-Free Processes, CONCUR-92, LNCS 630

[32] Rance Cleaveland, Matthew Hennessy, Priorities in Process Algebras, LICS 1988.

[33] Rance Cleaveland, Joachim Parrow, and Bernard Steffen. *The Concurrency Workbench.* 1988. *TOPLAS 15(1), 1993*

[34] R. de Nicola and M. Hennessy. *Testing Equivalences for Processes. Theoretical Computer Science*, 34:83–133, 1984.

[35] Rocco de Nicola and Matthew Hennessy. *CCS without τ's.* In *LNCS 249: TAPSOFT '87*, pages 138–152, Springer-Verlag, Berlin, 1987.

[36] P. Degano, R. de Nicola, and U. Montanari. *A Distributed Operational Semantics for CCS based on Condition/Event Systems.* Technical Report Nota Interna I.E.I B4-21, Department of Computer Science, University of Pisa, 1987. To appear in Acta Informatica.

[37] R. DeNicola, F. W. Vaandrager, Three logics for branching bisimulation, LICS 1990

[38] Dijkstra, E. W., Guarded commands, nondeterminacy and the formal derivation of programs, *Comm. of the ACM*, 18(8):453–457, August 1975.

[39] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, On-the-fly Garbage Collection: An Exercise in Cooperation, Comm. ACM 21(11), 1978, pp966-975

[40] Edsger W. Dijkstra, *Selected Writings on Computing, A personal perspective* , Springer Verlag, 1982

[41] H. Ehrig, B. Mahr, Fundamentals of Algebraic Specification 1, Springer Verlag, 1985

[42] Tzilla Elrad and Nissim Francez, A weakest precondition semantics for communicating processes, RC9773, IBM Yorktown Heights, 1982

[43] U. Engberg and M. Nielsen. *A Calculus of Communicating Systems with Label Passing.* Technical Report DAIMI PB-208, Aarhus University Computer Science Department, 1986. The original paper in label passing CCS that inspired the π-calculus of Milner, Parrow and Walker.

[44] Chris George, The RAISE Specification Language, A Tutorial, in VDM '91 Formal Software Development Methods, Vol 2: Tutorials, LNCS 552

[45] Rob Gerth, A sound and complete Hoare axiomatization of the Ada rendezvous, in Proc. 9th ICALP, LNCS 140, pp252–264, 1982

[46] R. Gerth and W. P. de Roever. *A proof system for concurrent Ada programs. Science of Computer Programming*, 4, 198.

[47] R. J. van Glabbeek, W. P. Weijland, Banching time and abstraction in bisimulation semantics, in Information Processing 89, pp 613–618

[48] M. Gordon et al, The HOL System Description, Cambridge University and SRI International

[49] S. Graf and J. Sifakis. *A modal characterization of observational congruence of finite terms of CCS. Information and Control*, 68:125–145, 1986.

[50] David Gries, *The Science of Programming*, Springer Verlag, 1981

[51] C. A. Gunter, D. S. Scott, Semantic Domains, in Handbook of Theoretical Computer Science (J. van Leeuwen ed.), The MIT Press, 1990

[52] D. Harel, First-Order Dynamic Logic, LNCS 68, 1979

[53] Matthew Hennessy, Algebraic Theory of Processes, MIT Press, 1988

[54] Matthew Hennessy, Proving Systolic Systems Correct, TOPLAS 8(3), pp344-387, 1986

[55] M. Hennessy and R. Milner. *Algebraic Laws for Nondeterminism and Concurrency. Journal of the ACM*, 32(1):137–161, 1985.

[56] Sooner is Safer Than Later, TR 92-1309, Cornell University, 1992

[57] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985

[58] Soren Holmström. *Hennessy-Milner Logic with Recursion as a Specification Language, and a Refinement Calculus based on it.* Programming Methodology Group Report 44, University of Göteborg, 1988.

[59] A. Ingolfsdottir, B. Thomsen, Semantic Models for CCS with Values, Chalmers Workshop on Concurrency, Baastad, Sweden, 1991

[60] Paola Inverardi, Corrado Priami, Evaluation of Tools for the Analysis of Communicating Systems, Bulletin of the EATCS 35, October 1991, pp158-185

[61] *occam Programming Manual.* 1984. Prentice-Hall Series in Computer Science, C. A. R. Hoare (Series Editor). See also the *occam 2 Reference Manual*, Prentice-Hall 1988.

[62] Radha Jagadeesan and Prakash Panangaden. *A Domain-Theoretic Model for a Higher-Order Process Calculus.* In M. S. Paterson, editor, *LNCS 443: Automata, Languages and Programming: Proceedings of the* 17^{th} *International Colloquium*, pages 181–194, Springer-Verlag, Berlin, July 1990. Full form issued as Cornell technical report 89-1058.

[63] C. T. Jensen, The Concurrency Workbench with Priorities, 3rd Int'l Workshop Computer Aided Verification 1991, LNCS 575

[64] P. C. Kanellakis and S. A. Smolka. *CCS Expressions, Finite State Processes, and Three Problems of Equivalence.* In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing, Montreal, Canada*, pages 228–240, August 1983.

[65] P. C. Kanellakis and S. A. Smolka. *On the Analysis of Cooperation and Antagonism in Networks of Communicating Processes. Algorithmica*, 1988.

[66] L. Kossen, W. P. Weijland, Correctness Proofs for Systolic Algorithms: Palindromes and Sorting, Report FVI 87-04, Comp. Sci. Dept., University of Amsterdam, 1987

[67] D. Kozen. *Results on the propositional mu-calculus.* In *LNCS 140: Proceedings of the 9th International Colloqium on Automata, Languages and Programming (ICALP)*, Springer-Verlag, Berlin, 1982.

[68] Saul A. Kripke, Semantic Considerations on Modal Logic, in Reference and Modality (Leonard Linsky, ed.), Oxford University Press, 1971, pp 63-72

[69] Rom Langerak, Transformation and Semantics for LOTOS, Twente University, 1992

[70] Lankford, D. S., Canonical algebraic simplification in computational logic, Memo ATP-25, Automatic Theorem Proving Project, University of Texas, Austin, 1975.

[71] L. Lamport, The 'Hoare Logic' of Concurrent Programs, Acta Informatica 14, pp. 21–37, 1980

[72] L. Lamport, Specifying Concurrent Program Modules, TOPLAS, 5/2, pp. 190–222, April 1983

[73] K. G. Larsen and R. Milner. *Verifying a protocol using relativized bisimulation.* In *LNCS 267: Proceedings of the 14th International Colloqium on Automata, Languages and Programming (ICALP)*, Springer-Verlag, Berlin, 1987.

[74] K. G. Larsen and A. Skou. *Bisimulation through Probabilistic Testing.* Technical Report R 88-29, Institut for Elektronsiske Systemer, Afdeling for Matematik og Datalogi, Aalborg Universitetscenter, 1988. Full form of POPL '89 paper.

[75] Kim Larsen. *Proof systems for Hennessy-Milner logic with recursion.* In *LNCS 299: Proceedings of CAAP 1988*, Springer-Verlag, Berlin, 1988. Full version to appear in TCS.

[76] Kim G. Larsen. *Context-Dependent Bisimulation between Processes.* PhD thesis, University of Edinburgh, 1986. Keywords: Contexts as predicate transformers. Modal Logic, bisimulation.

[77] Huimin Lin, PAM: A Process Algebra Manipulator, 3rd Int'l Workshop Computer Aided Verification, 1991, LNSC 575

[78] Michael G. Main, A powerdomain primer, Bulletin of the EATCS 33, October 1987

[79] Michael G. Main, Free Constructions of Powerdomains, LNCS 239, pp162–183

[80] Sigurd Meldal, On Hierarchical Abstraction and Partial Correctness of Concurrent Structures, PhD Thesis, University of Oslo, May 1986

[81] S. Mau and G. J. Veltink. *A Process Specification Formalism.* Programming Research Group Report P8814, University of Amsterdam, 1988.

[82] S. Mau, G. J. Veltink, An Introduction to PSF_d, Proc, Int. Joint Conf. on Theory and Practice of Software Development, LNSC 352, pp272-285, 1989

[83] George J. Milne, Robin Milner, Concurrent Processes and Their Syntax, Journal of the ACM 26/2, pp. 302–321, April 1979

[84] Robert Milne, Christopher Strachey, A theory of programming language semantics, Chapman and Hall, 1976

[85] Robin Milner. *Communication and Concurrency.* Series in Computer Science, Prentice-Hall, 1989.

[86] Robin Milner, Flowgraphs and Flow Algebras, Journal of the ACM 26/4, pp. 794–818, October 1979

[87] R. Milner, *A calculus of Communicating Systems*, LNCS 92, 1980

[88] R. Milner, Operational and Algebraic Semantics of Concurrent Processes, in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), Elsevier, 1990

[89] Robin Milner. *Calculi for Synchrony and Asynchrony. Theoretical Computer Science,* 25:269–310, 1983.

[90] Robin Milner. *Lectures on a Calculus for Communicating Systems.* In *LNCS 197: Proceedings of the Seminar on Concurrency*, pages 197–220, Springer-Verlag, Berlin, 1985.

[91] R. Milner, A complete inference system for a class of regular behaviours. Journal of Computer and System sciences 12 (3), pp439–466, 1984

[92] Robin Milner. *Operational and Algebraic Semantics of Concurrent Processes.* Internal Report ECS-LFCS-88-46, University of Edinburgh, 1988.

[93] Robin Milner, Joachim Parrow, and David Walker. *A Calculus of Mobile Processes.* Technical Report ECS-LFCS-89-86, LFCS, Department of Computer Science, University of Edinburgh, June 1989. Also published as CSR-303-89.

[94] Robin Milner, Joachim Parrow, and David Walker. *Modal Logics for Mobile Processes.* In *Proceedings of Concur '91, LNCS 527*, 1991.

[95] Faron Moller. *Axioms for Concurrency.* PhD thesis, University of Edinburgh, 1989. Report Number ECS-LFCS-89-84.

[96] M. Nezi, Mechanising a Proof by Induction of Process Algebra Specifications in Higher Order Logic, rd3 Int'l Workshop Computer Aided Verification, 1991, LNCS 575.

[97] R. De Nicola, P. Inverardi, M. Nesi, Using the Axiomatic Representation of Behavioural Equivalences for Manipulating CCS Specifications, LNCS 407, 1990

[98] O. Nierstrasz, M. Papathomas, Viewing Objects as Patterns of Communicating Agents, Proceedings on the Conference on Object-Oriented Programming, Ottawa 1990, pp 38–43 (also, SIGPLAN Notices 25(10), October 1990)

[99] E.-R. Olderog and C. A. R. Hoare. *Specification-oriented Semantics for Communicating Processes. Acta Informatica*, 23:9–66, 1986. Full form of LNCS 154: 10th ICALP paper.

[100] Owicki, S., Gires, D., An axiomatic proof technique for parallel programs, Acta Informatica 6, pp 319-340, 1979.

[101] D. Park. *Concurrency and Automata on Infinite Sequences.* In *Proceedings of the* 5^{th} *G.I. Conference*, Springer-Verlag, Berlin, 1981.

[102] I. Phillips. *Refusal Testing.* In *LNCS 226: Automata, Languages and Programming: Proceedings of the* 13^{th} *International Colloquium*, pages 304–313, Springer-Verlag, Berlin, 1986.

[103] G. D. Plotkin. *A Powerdomain Construction. SIAM Journal of Computing*, 5:452–486, 1976.

[104] G. D. Plotkin. *A Structural Approach to Operational Semantics.* Technical Report DAIMI FN-19, Aarhus University, September 1981.

[105] G. D. Plotkin. *An Operational Semantics for CSP.* In D. Bjørner, editor, *Formal Description of Programming Language Concepts II*, pages 199–223, North-Holland, Amsterdam, 1983.

[106] A. Pnueli. *Linear and Branching Structures in the Semantics and Logics of Reactive Systems.* In *LNCS 194: Automata, Languages and Programming: Proceedings of the* 12^{th} *International Colloquium*, pages 15–32, Springer-Verlag, Berlin, 1985.

[107] W. Polak, Program Verification Based on Denotational Semantics, *Eighth annual ACM Symposium on Principles of Programming Languages*, Williamsburg, January 1981, pp 149–158

[108] W. Polak, Formal verification of Ada tasking programs, ORA proposal, Oct. 1988.

[109] W. Polak, Predicate Transformer Semantics for Ada, ORA internal report.

[110] W. Polak, Report on subset for concurrency, STARSSC03094/001/00, 1990.

[111] W. Polak, Predicate Transformer Semantics for Ada tasking, ORA internal report. Dec. 1991

[112] Ichiro Satoh, Mario Tokoro, A formalism for real-time concurrent Object-Oriented Computing OOPSLA, 1992, pp. 315–326

[113] M. B. Smyth. *Powerdomains. JCSS*, 16:23–36, 1978.

[114] Colin Stirling. *Modal Logics for Communicating Systems. Theoretical Computer Science*, 49:311–347, 1987.

[115] Colin Stirling. *Temporal Logics for CCS.* In *Proceedings of REX Workshop 1988*, Springer-Verlag, Berlin, 1988.

[116] N. Soundarajan, Axiomatic Semantics of Communicating Sequential Processes, ACM TOPLAS, 6/4, pp. 647–662, October 1984

[117] Bent Thomsen. *Calculi for Higher-Order Communicating Systems.* PhD thesis, Imperial College of Science, Technology and Medicine, London, Department of Computing, 1990.

[118] Bent Thomsen. *A Calculus of Higher Order Communicating Systems.* In *Conference Record of the Sixteenth Annual ACM Symposium on Principle of Programming Languages*, pages 143–154, ACM, January 1989.

[119] R. J. van Glabbeek. *The semantics of finite, concrete, sequential processes.* Technical Report Report RvG-8801, Center for Mathematics and Computer Science, Amsterdam, 1988.

[120] F. W. Vaandrager, Verification of Two Communication Protocols by means of Process Algebra, CS-R8608, CWI, Amsterdam, 1986

[121] Philip Wadler, Comprehending Monads, Math. Struct. in Comp. Science (1992), Vol 2, pp. 461–493.

[122] David J. Walker. *Bisimulation and Divergence. Information and Computation*, 85(2):202–241, 1990.

[123] J. Zwiers. *Predicates, Predicate Transformers and Refinement.* In *LNCS 430: Stepwise Refinement of Distributed Systems. (Proceedings of the 1989 REX Workshop)*, pages 760–776, Springer-Verlag, Berlin, 1989.

[124] J. Zwiers, W. P. de Roever, and P. van Emde Boas. *Compositionality and concurrent networks: soundness and completeness of a proof system.* In *LNCS 194: Proceedings of the 12th International Colloqium on Automata, Languages and Programming (ICALP)*, pages 509–519, Springer-Verlag, Berlin, 1985.